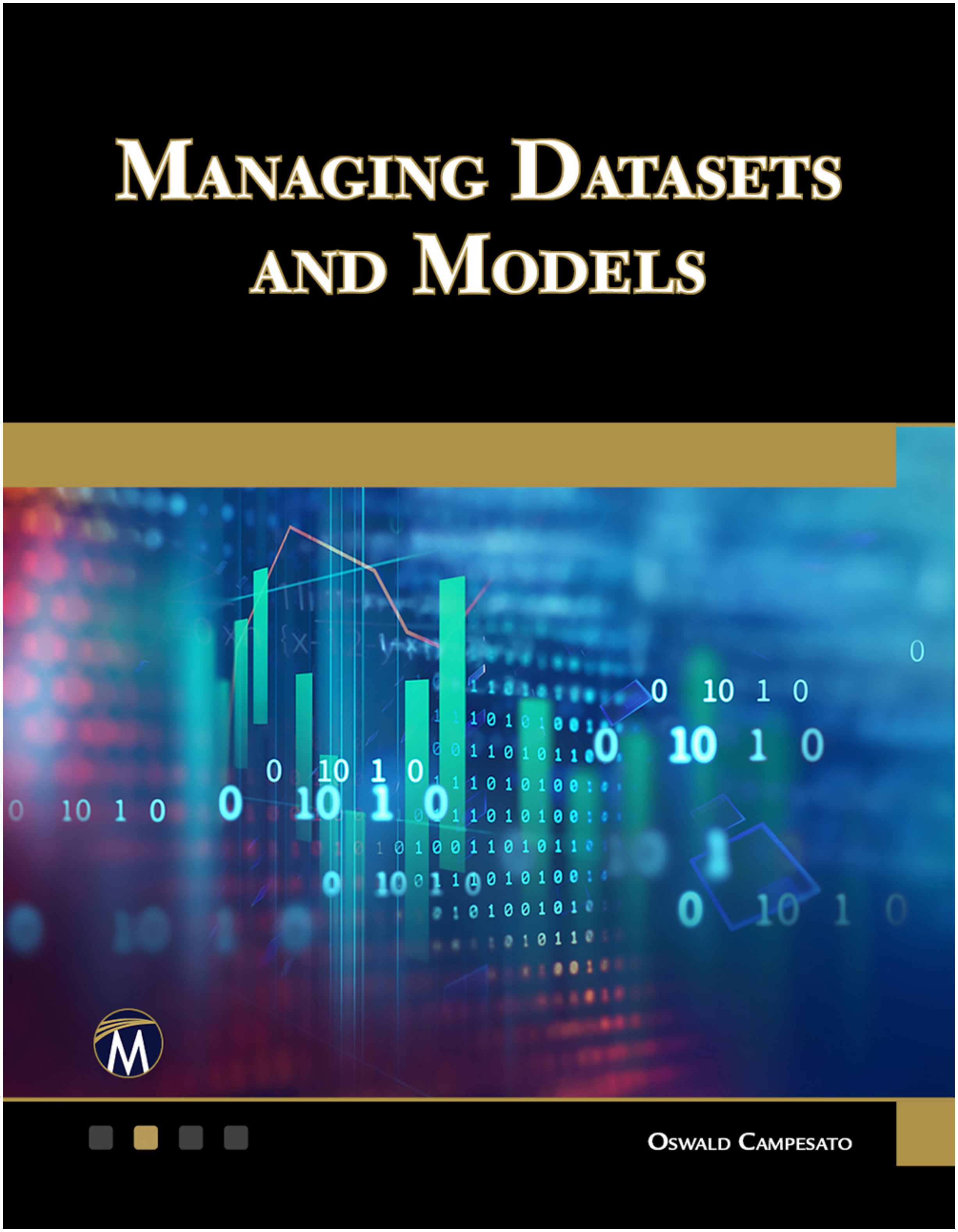
MANAGING DATASETS AND MODELS

OSWALD CAMPESATO



MANAGING DATASETS AND MODELS

OSWALD CAMPESATO

OceanofPDF.com

OceanofPDF.com

MANAGING DATASETS AND MODELS

OceanofPDF.com

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and its companion files (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information, files, or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

MERCURY LEARNING AND INFORMATION ("MLI" or "the Publisher") and anyone involved in the creation, writing, production, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and only at the discretion of the Publisher. The use of “implied warranty” and certain “exclusions” vary from state to state, and might not apply to the purchaser of this product.

Companion files also available for downloading from the publisher by writing to info@merclearning.com.

OceanofPDF.com

MANAGING DATASETS
AND
MODELS

Oswald Campesato

MERCURY LEARNING AND INFORMATION
Dulles, *Virginia*
Boston, Massachusetts
New Delhi

OceanofPDF.com

Copyright ©2023 by MERCURY LEARNING AND INFORMATION LLC. All rights reserved.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai
MERCURY LEARNING AND INFORMATION
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
1-800-232-0223

O. Campesato. *Managing Datasets and Models*.
ISBN: 9781683929529

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2022952302

232425321 Printed on acid-free paper in the United States of America.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are also available in digital format at numerous digital vendors. *Companion files are available for download by writing to the publisher at info@merclearning.com*. The sole obligation of MERCURY LEARNING AND INFORMATION to the purchaser is to replace the book, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

OceanofPDF.com

I’d like to dedicate this book to my parents
– may this bring joy and happiness into their lives.

[*OceanofPDF.com*]

CONTENTS

Preface

Chapter 1: Working with Data

Import Statements for this Chapter
Exploratory Data Analysis (EDA)
Dealing with Data: What Can Go Wrong?
Analyzing Missing Data
Explanation of Data Types
Data Preprocessing
Working with Data Types
What is Drift?
What is Data Leakage?
Model Selection and Preparing Datasets
Types of Dependencies Among Features
Data Cleaning and Imputation
Summary

Chapter 2: Outlier and Anomaly Detection

Import Statements for this Chapter
Working with Outliers
Finding Outliers with NumPy
Finding Outliers with Pandas
Finding Outliers with Scikit-Learn (Optional)
Fraud Detection
Techniques for Anomaly Detection

Working with Imbalanced Datasets
Summary
Reference

Chapter 3: Cleaning Datasets
Prerequisites for this Chapter
Analyzing Missing Data
Pandas, CSV Files, and Missing Data
Missing Data and Imputation
Skewed Datasets
CSV Files with Multi-Row Records
Column Subset and Row Subrange of Titanic CSV File
Data Normalization
Handling Categorical Data
Working with Currency
Working with Dates
Working with Quoted Fields
What is SMOTE?
Data Wrangling
Summary

Chapter 4: Working with Models
Import Statements for this Chapter
Techniques for Scaling Data
Examples of Splitting and Scaling Data
The Confusion Matrix
The ROC Curve and AUC Curve
Exploring the Titanic Dataset
Steps for Training Classifiers
Diagram for Partitioned Datasets
A KNN-Based Model with the wine.csv Dataset

Other Models with the wine.csv Dataset
A KNN-Based Model with the bmi.csv Dataset
A KNN-Based Model with the Diabetes.csv Dataset
SMOTE and the Titanic Dataset
EDA and Data Visualization
What about Regression and Clustering?
Feature Importance
What is Feature Engineering?
What is Feature Selection?
What is Feature Extraction?
Data Cleaning and Machine Learning
Summary

Chapter 5: Matplotlib and Seaborn
Import Statements for this Chapter
What is Data Visualization?
What is Matplotlib?
Matplotlib Styles
Display Attribute Values
Color Values in Matplotlib
Cubed Numbers in Matplotlib
Horizontal Lines in Matplotlib
Slanted Lines in Matplotlib
Parallel Slanted Lines in Matplotlib
Lines and Labeled Vertices in Matplotlib
A Dotted Grid in Matplotlib
Lines in a Grid in Matplotlib
Two Lines and a Legend in Matplotlib
Loading Images in Matplotlib
A Checkerboard in Matplotlib
Randomized Data Points in Matplotlib

A Set of Line Segments in Matplotlib
Plotting Multiple Lines in Matplotlib
Trigonometric Functions in Matplotlib
A Histogram in Matplotlib
Histogram with Data from a Sqlite3 Table
Plot a Best-Fitting Line with ggplot
Plot Bar Charts
Plot a Pie Chart
Heat Maps
Save Plot as a PNG File
Working with SweetViz
Working with Skimpy
3D Charts in Matplotlib
Plotting Financial Data with Mplfinance
Charts and Graphs with Data from Sqlite3
Working with Seaborn
Seaborn Dataset Names
Seaborn Built-In Datasets
The Iris Dataset in Seaborn
The Titanic Dataset in Seaborn
Extracting Data from Titanic Dataset in Seaborn (1)
Extracting Data from Titanic Dataset in Seaborn (2)
Visualizing a Pandas Data Frame in Seaborn
Seaborn Heat Maps
Seaborn Pair Plots
What is Bokeh?
Introduction to Scikit-Learn
The Digits Dataset in Scikit-Learn
The Iris Dataset in Scikit-Learn (1)
The Iris Dataset in Scikit-Learn (2)

Advanced Topics in Seaborn
Summary

Appendix: Working with awk
The awk Command
Aligning Text with the printf() Statement
Conditional Logic and Control Statements
Deleting Alternate Lines in Datasets
Merging Lines in Datasets
Matching with Metacharacters and Character Sets
Printing Lines Using Conditional Logic
Splitting File Names with awk
Working with Postfix Arithmetic Operators
Numeric Functions in awk
One-Line awk Commands
Useful Short awk Scripts
Printing the Words in a Text String in awk
Count Occurrences of a String in Specific Rows
Printing a String in a Fixed Number of Columns
Printing a Dataset in a Fixed Number of Columns
Aligning Columns in Datasets
Aligning Columns and Multiple Rows in Datasets
Removing a Column from a Text File
Subsets of Column-Aligned Rows in Datasets
Counting Word Frequency in Datasets
Displaying Only “Pure” Words in a Dataset
Working with Multi-Line Records in awk
A Simple Use Case
Another Use Case
Summary

Index

OceanofPDF.com

PREFACE

WHAT IS THE PRIMARY VALUE PROPOSITION FOR THIS BOOK?

This book contains a fast-paced introduction to data-related tasks in preparation for training models on datasets. Keep in mind that this book presents the necessary sequence of steps in order to train models on classification tasks. You will see a detailed (i.e., step-by-step) `Python`-based code sample that uses the `kNN` algorithm to train a model on a dataset.

Next, you will see other classification algorithms (on the same dataset), such as decision trees, random forests, SVMs (support vector machines), and Naive Bayes *simply by modifying three lines of code*.

As a quick overview, Chapter 1 starts with an introduction to datasets and issues that can arise, followed by Chapter 2 on outliers and anomaly detection. Chapter 3 explores ways for handling missing data and invalid data, and Chapter 4 shows you how to train models with classification algorithms. In particular, the section called “Steps for Training Classifiers” explains the required sequence of steps, along with a code sample that implements those steps. Chapter 5 introduces visualization toolkits, such as Sweetviz, Skimpy, Matplotlib, and Seaborn, along with some simple Python-based code samples that render charts and graphs. The Appendix introduces the awk utility.

Again, keep in mind that the details regarding the design or implementation of classification algorithms are outside the scope of this book, but you can find online tutorials that explain how they work.

WHAT DO I NEED TO KNOW FOR THIS BOOK?

The minimum programming requirement is a basic knowledge of `Python` 3.x because all the code samples are in `Python`. In some cases, you need a rudimentary understanding of the `awk` utility, which you can learn through free online tutorials.

In addition, you need ta basic understanding of Pandas data frames and the Pandas methods for extracting information from data frames.

ARE ALL CODE SAMPLES COMPLETE?

Although the code samples are complete, keep in mind that sometimes it might be necessary to use pip to install Python modules that are referenced in the Python code samples, but are not installed yet on your laptop.

Moreover, several code samples are written in `awk`, which is a command line utility (part of UNIX and Linux), and one code sample in the appendix is a `Java` program. Keep in mind that `Java` is not covered in this book: treat the Java code samples as optional or read some online tutorials regarding Java. Note that the awk utility is introduced in the Appendix.

WHAT WILL I LEARN FROM THIS BOOK?

The introductory section of this preface contains a brief outline of the topics in each of the chapters of this book. As the title suggests, you will acquire a solid understanding of the statistical concepts that you will encounter as a data scientist.

Moreover, you will be exposed to concepts and statistical tests that could prove useful later in your career, even if they are not needed at this stage in your career as a data scientist.

THE TARGET AUDIENCE

This book is intended for people who have limited experience in managing datasets in machine learning. This book is also intended to reach an international audience of readers with highly diverse backgrounds. While many readers know how to read English, their native spoken language is not English (which could be their second, third, or even fourth language). Consequently, this book uses standard English rather than colloquial expressions in order to maximize clarity.

GETTING THE MOST FROM THIS BOOK

Some programmers learn well from prose, others learn well from sample code (and lots of it), which means that there's no single style that can be used for everyone.

Moreover, some programmers want to run the code first, see what it does, and then return to the code to delve into the details (and others use the opposite approach).

Consequently, there are various types of code samples in this book: some are short, some are long, and other code samples "build" from earlier code samples.

DOES THIS BOOK CONTAIN PRODUCTION-LEVEL CODE SAMPLES?

The primary purpose of the code samples in this book is to show you how to solve tasks that arise when you train models with classification algorithms. Hence, clarity has higher priority than writing more compact or highly optimized code; for example, inspect the loops in the Python code sample to see if they can be made more efficient. Suggestion: treat such code samples as opportunities for you to optimize the code samples in this book.

If you decide to use any of the code in this book in a production environment, you ought to subject that code to the same rigorous analysis as the other parts of your code base.

WHAT ARE THE NON-TECHNICAL PREREQUISITES FOR THIS BOOK?

Although the answer to this question is more difficult to quantify, it's very important to have strong desire to learn about statistical concepts, along with the motivation and discipline to read and understand the code samples (and ideally enhance the contents of the code samples).

HOW DO I SET UP A COMMAND SHELL?

If you are a Mac user, there are three ways to do so. The first method is to use `Finder` to navigate to `Applications > Utilities` and then double click on the `Utilities` application. Next, if you already have a command shell available, you can launch a new command shell by typing the following command:

```
open /Applications/Utilities/Terminal.app
```

A second method for Mac users is to open a new command shell on a MacBook from a command shell that is already visible simply by clicking `command+n` in that command shell, and your Mac will launch another command shell.

If you are a PC user, you can install Cygwin (open source *https://cygwin.com/*) that simulates bash commands or use another toolkit such as MKS (a commercial product). Please read the online documentation that describes the download and installation process. Note that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as .bash_login).

COMPANION FILES

All of the code samples and figures in this book may be obtained by writing to the publisher at info@merclearning.com.

WHAT ARE THE "NEXT STEPS" AFTER FINISHING THIS BOOK?

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. If you are interested primarily in learning more about machine learning, then this book is a "steppingstone" to other books that contain more complex datasets as well as code samples for linear regression and clustering tasks.

If you want to explore other areas of machine learning, there are some subfields of machine learning, such as deep learning and reinforcement learning (and deep reinforcement learning) which might appeal to you. Fortunately, there are many resources available, and you can perform an

Internet search for those resources. One other point: the aspects of machine learning for you to learn depend on who you are: the needs of a data scientist, machine learning engineer, development manager, software developer, or student are all different.

O. Campesato
February 2023

OceanofPDF.com

CHAPTER 1

WORKING WITH DATA

This chapter focuses on data types that you will encounter in datasets, including currency and dates, as well as scaling data values and various aspects of feature engineering. This chapter starts with simple material and rapidly progresses to machine learning concepts, which include a code sample that uses the `DecisionTreeClassifier` class from scikit-learn to determine feature importance in a dataset. This book is intended for data scientists, which means you are familiar with decision trees, and any new topics that you encounter in this chapter are within your grasp.

The first section of this chapter briefly discusses some aspects of EDA (Exploratory Data Analysis), such as data quality and the data-centric AI versus model-centric AI, as well as some of the steps involved in data cleaning and data wrangling. You will also see an EDA code sample involving the Titanic dataset.

The second section of this chapter describes common types of data, such as binary, nominal, ordinal, and categorical data. In addition, you will learn about continuous versus discrete data, quantitative and quantitative data, and types of statistical data.

The third section introduces the notion of data drift and data leakage, followed by model selection. This section also describes how to process categorical data, and how to map categorical data to numeric data.

The fourth section discusses concepts such as homoskedasticity, collinearity, variance inflation factor, and correlation. This section contains Python-based code samples that involve currency and date values. You will also learn about various aspects of splitting datasets and scaling data values.

The fifth section introduces feature engineering, feature selection, and feature extraction, and also discusses how they differ from each other. You will also learn about feature scaling in some machine learning algorithms, labeled versus unlabeled data, and training large datasets.

NOTE: This chapter contains shell scripts that use basic features of awk and Pandas, which you can learn about from online tutorials.

IMPORT STATEMENTS FOR THIS CHAPTER

This chapter contains a mixture of Python-based code samples, an awk-based shell script, and a Java code sample to show you how to solve tasks using different technologies. All the code samples are straightforward, and if you can follow the Pandas and awk-based code samples in this chapter, then you will most likely be able to understand the awk-based code samples in subsequent chapters.

This chapter requires basic knowledge of Python and Pandas, such as creating Pandas data frames, as well as reading and writing comma separated values (`CSV`) files. Knowledge of the awk programming language is required for shell scripts that invoke the `awk` command, if you decide you want to read those code samples.

In addition, the following list contains all the `import` statements that you will encounter in the Python code samples for this chapter:

```
from scipy import stats
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_classification
from sklearn.ensemble import
RandomForestClassifier
from sklearn.feature_selection import RFE
from sklearn.model_selection import
train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import sys
```

EXPLORATORY DATA ANALYSIS (EDA)

According to Wikipedia, EDA involves analyzing datasets to summarize their main characteristics, often with visual methods. EDA also involves searching through data to detect patterns (if there are any) and anomalies, and in some cases, test hypotheses regarding the distribution of the data.

EDA represents the initial phase of data analysis, whereby data is explored to determine its primary characteristics. Moreover, this phase involves detecting patterns (if any) and outstanding issues pertaining to the data. The purpose of EDA is to obtain an understanding of the semantics of the data without performing a deep assessment of the nature of the data.

The analysis is often performed through data visualization to produce a summary of their most important characteristics. The four types of EDA are listed here:

- univariate non-graphical
- multivariate non-graphical
- univariate graphical
- multivariate graphical

In brief, the two primary methods for data analysis are *qualitative data analysis* techniques and *quantitative data analysis* techniques.

As an example of exploratory data analysis, consider the plethora of cell phones that customers can purchase for various needs (work, home, and minors). Visualize the data in an associated dataset to determine the top ten (or top three) most popular cell phones, which can potentially be performed by state (or province) and country.

An example of quantitative data analysis involves measuring (quantifying) data, which can be gathered from physical devices, surveys, or activities such as downloading applications from a webpage.

Common visualization techniques used in EDA include histograms, line graphs, bar charts, box plots, and multi-variate charts.

What is Data Quality?

According to Wikipedia, *data quality* refers to “the state of qualitative or quantitative pieces of information.” Furthermore, high data quality refers to data whose quality meets the various needs of an organization. In particular, data cleaning tasks are the type of tasks that assist in achieving high data quality.

When companies label their data, they obviously strive for a high quality of labeled data, and yet the quality can be adversely affected in various ways, some of which are listed here:

- inaccurate methodology for labeling data
- insufficient data accuracy
- insufficient attention to data management

The cumulative effect of the preceding (and other) types of errors can be significant, to the extent that models underperform in a production environment. In addition to the technical aspects, underperforming models can have an adverse effect on business revenue.

Related to data quality is *data quality assurance,* which typically involves data cleaning tasks that are discussed later in this chapter, after which data is analyzed to detect potential inconsistencies in the data, and then determine how to resolve those inconsistencies. Another aspect to consider: the aggregation of additional data sources, especially involving heterogenous sources of data, can introduce challenges with respect to ensuring data quality. Other concepts related to data quality include *data stewardship* and *data governance,* both of which are discussed in multiple online articles.

Data-Centric AI or Model-Centric AI?

A *model-centric* approach focuses primarily on enhancing the performance of a given model, and data is considered secondary in importance. In fact, during the past ten years or so, the emphasis of AI has been a model-centric approach. Note that during this time span, some very powerful models and architectures have been developed, such as the Convolutional Neural Network (`CNN`) model for image classification in 2012 and the enormous impact (especially in Natural Language Processing, `NLP`) of models based on the transformer architecture that was developed in 2017.

By contrast, a *data-centric* approach concentrates on improving data, and it relies on several factors, such as the quality of labels for the data as well as obtaining accurate data for training a model.

Given the importance of high quality data with respect to training a model, it stands to reason that using a data-centric approach instead of a model-centric approach can result in higher quality models in AI. While data quality and model effectiveness are both important, the data-centric approach is becoming increasingly more strategic in the machine learning world. For more information, visit the following site:

https://research.aimultiple.com/data-centric-ai/

The Data Cleaning and Data Wrangling Steps

The next step often involves *data cleaning* to find and correct errors in the dataset, such as missing data, duplicate data, or invalid data. This task also involves data consistency, which pertains to updating different representations of the same value in a consistent manner. As a simple example, suppose that a webpage contains a form with an input field whose valid input is either `Y` or `N`, but users are able to enter `Yes`, `Ys`, or `ys` as text input. Obviously, these values correspond to the value `Y`, and they must all be converted to the value `Y` to achieve data consistency.

Finally, *data wrangling* can be performed after the data cleaning task is completed. Although interpretations of data wrangling do vary, in this book the term refers to transforming datasets into different formats as well as combining two or more datasets. Hence, data wrangling does not examine the individual data values to determine whether they are valid: this step is performed during data cleaning.

Sometimes it is worthwhile to perform another data cleaning step after the data wrangling step. For example, suppose that two CSV files contain employee-related data, and you merge these CSV files into a third CSV file. The newly created CSV file might contain duplicate values: it is certainly possible to have two people with the same name (such as John Smith).

ELT and ETL

ELT is an acronym for extract, load, and transform, which is a pipeline-based approach for managing data. Another pipeline-based approach is called ETL (extract, transform, and load), which is more popular than `ELT`. However, `ELT` has the following advantages over `ETL`:

- ELT requires less computational time.
- ELT is well-suit for processing large datasets.
- ELT is more cost-effective.

`ELT` is a process that extracts, loads, and transforms data from one or more sources to a data warehouse or other unified data repository. `ELT` is a data integration process that is similar to its counterpart `ETL`.

DEALING WITH DATA: WHAT CAN GO WRONG?

In a perfect world, all datasets are in pristine condition, with no extreme values, no missing values, and no erroneous values. Every feature value is captured correctly, with no chance for any confusion. Moreover, no conversion is required between date formats, currency values, or languages because of the "One Universal Standard" that defines the correct formats and acceptable values for every possible set of data values.

Of course, all the scenarios in the previous paragraph can and do occur, which is the reason for the techniques that are discussed in this chapter.

Even after you manage to create a wonderfully clean and robust dataset, other issues can arise, such as data drift, which is described in a later section.

In fact, the task of cleaning data is not necessarily complete even after a machine learning model is deployed to a production environment. For instance, an online system that gathers terabytes or petabytes of data on a daily basis can contain skewed values that adversely affect the performance of the model. Such adverse affects can be revealed through the changes in the metrics that are associated with the production model.

Datasets

In general, a *dataset* is a data source (such as a text file) that often contains rows and columns of data. Each row is typically called a *data point*, and each column is called a *feature*. A dataset can be a CSV (comma separated values), TSV (tab separated values), Excel spreadsheet, a table in an RDBMS, a document in a NoSQL database, the output from a Web Service, and so forth.

Note that a *static dataset* consists of fixed data. For example, a CSV file that contains the states of the USA is a static dataset. A slightly different example involves a product table that contains information about the products that customers can buy from a company. Such a table is static if no new products are added to the table. Discontinued products are probably maintained as historical data that can appear in product-related reports.

By contrast, a *dynamic dataset* consist of data that changes over a period of time. Simple examples include housing prices, stock prices, and time-based data from Internet of Things (IoT) devices.

A dataset can vary from very small (perhaps a few features and 100 rows) to very large (more than 1,000 features and more than one million rows). If you are unfamiliar with the problem domain for a particular dataset, then you might struggle to determine its most important features. In this situation, you ought to consult a domain expert who understands the importance of the features, their inter-dependencies (if any), and whether the data values for the features are valid. In addition, there are algorithms (called dimensionality reduction algorithms) that can help you determine the most important features, such as PCA (Principal Component Analysis).

Before delving into topics such as data preprocessing and data types, let’s take a brief detour to introduce the concept of feature importance, which is discussed in greater detail in Chapter 4.

Someone needs to analyze the dataset to determine which features are the most important and which features can be safely ignored to train a model with the given dataset. A dataset can contain various data types, such as

- audio data
- image data
- numeric data
- text-based data
- video data
- combinations of the above

In this book, we will only consider datasets that contain columns with numeric or text-based data types, which can be further classified as follows:

- nominal (string-based or numeric)
- ordinal (ordered values)
- categorical (enumeration)
- interval (positive/negative values)
- ratio (non-negative values)

A subsequent section briefly describes the data types that are in the preceding bullet list. For more information, please visit the following site:

https://careerfoundry.com/en/blog/data-analytics/what-is-ordinal-data/

ANALYZING MISSING DATA

This section contains subsections that describes types of missing data, common causes of missing data, and various ways to input values for missing data. Outlier detection, fraud detection, and anomaly detection pertain to *existing* data that is problematic, with varying degrees of severity.

By contrast, missing data presents a different issue by its very absence, which raises the following question: what can you do about the missing values? Is it better to discard data points (e.g., rows in a CSV file) with missing values, or is it better to estimate reasonable values as a replacement for the missing values? Missing data can adversely affect a thorough analysis of a dataset, whereas erroneous data can increase bias and uncertainty.

At this point, you have undoubtedly realized that a single solution does not exist for every dataset: you need to perform an analysis on a case-by-case basis, after you have learned some of the techniques that might help you effectively address missing data values.

Classifying Missing Data: MCAR, MAR, and MNAR

Donald Rubin proposed a classification system for missing data based on whether such missing data is due to random factors. According to his classification system, missing data can belong to one the following categories:

1. MCAR (Missing Completely At Random)

2. MAR (Missing At Random)
3. MNAR (Missing Not At Random)

Type #1 suggests that missing data is due to factors that are unrelated to the data. Type #2 applies to when there is equal probability of missing data that occurs within groups of the defined data. Type #3 refers to data (outside of Type #1 or Type #2) that is missing due to unknown reasons.

Causes of Missing Data

There are various reasons for missing values in a dataset, some of which are listed here:

- values are unavailable
- values were improperly collected
- inaccurate data entry

Although you might be tempted to *always* replace a missing values with a concrete value, there are situations in which you cannot determine a value. As a simple example, a survey that contains questions for people under 30 will have a missing value for respondents who are over 30, and in this case, specifying a value for the missing value would be incorrect. With these details in mind, there are various ways to fill missing values, some of which are listed here:

- remove row with a high percentage of missing values (50% or larger)
- one-hot encoding for categorical data
- handling data inconsistency
- use the Imputer class from the scikit-learn library
- fill missing values with the value in an adjacent row
- replace missing data with the mean/median/mode value
- infer ("impute") missing data values via `SMOTE`
- delete rows with missing data

Once again, the technique that you select for filling missing values is influenced by various factors, such as

- how you want to process the data
- the type of data involved
- the cause of missing values (see above)

Although the most common technique involves the mean value for numeric features, someone needs to determine which technique is appropriate for a given feature.

However, if you are not confident that you can impute a reasonable value, consider deleting the row with a missing value, and then train a model with the imputed value and also with the deleted row.

One problem that can arise after removing rows with missing values is that the resulting dataset is too small. In this case, consider using SMOTE (Synthetic Minority Oversampling Technique), which generates synthetic data.

EXPLANATION OF DATA TYPES

This section contains subsections that provide brief descriptions about the following data types:

- binary data
- nominal data
- ordinal data
- categorical data
- interval data
- ratio data

Later, you will learn about the difference between continuous data versus discrete data, as well as the difference between qualitative data versus quantitative data. In addition, the Pandas documentation describes data types and how to use them in Python.

Binary Data

Binary data involves data that can only take two distinct values. As such, binary data is the simplest type of data. A common example involves flipping a coin: the only outcomes are in the set {`H,T`}. Other terms for binary data include dichotomous, logical data, Boolean data, and indicator data. Binary data is also a type of categorical data that is discussed later.

Nominal Data

The word "nominal" has multiple meanings, and in our case, it refers to something that constitutes a name (the prefix "nom" means "name"). Thus, *nominal data* is often (see next paragraph) name-based data that involves different name labels. Examples of nominal data include hair color, music preferences, and movie types. As you can see, there is no hierarchy or ordering involved, so all values have the same importance. However, the number of items in nominal values might be different, such as the number of people who belong to different political parties.

Nominal data can involve numeric values to represent different values of a feature. For example, the numbers in the set {`0,1`} can represent {`Male, Female`}, and the numbers in the set {`0,1,2,3,4,5,6`} can represent the days in a week. However, there is no hierarchical interpretation associated with these numeric values: the day of the week represented by "0" is not considered more important or more valuable than any of the other numeric labels for the other days of the week. Instead, think of each element in terms of its predecessor and successor: note that the first element has no predecessor and the last element has no successor.

If you are familiar with programming languages, the counterpart to integer-based nominal values would be an enumeration, an example of which is here:

```
enum DAY {SUN,MON,TUE,WED,THU,FRI,SAT};
```

Ordinal Data

Ordinal data implies an ordering of the elements in a finite set (think "ordering" from the prefix "ord"). For example, there are different values for titles regarding software developers. As a simplified example, the set consisting of `{D1, D2, SD1, SD2}` can be used to specify junior developers (D1) up through senior developers (SD2), which have criteria associated with each level. Hence, integer-based and string-based elements of ordinal data are ordered.

Integer-based ordinal data does not have an implied relative value. For example, consider the following set of ordinal data `S = {1,2,3,4,5,6}` that represent grade levels in an organization. A level 2 employee is not "twice" as experienced as a level 1 employee, nor would a level 6 employee be three times as experienced as a level 2 (unless you define these values in such a manner).

Please read the `scikit-learn` documentation regarding the class `OrdinalEncoder` (`scikit-learn.preprocessing.OrdinalEncorder`) for handling ordinal data.

Categorical Data

Categorical data refers to nominal data as well as ordinal data: please read the preceding sections regarding the nuances involved in nominal data and ordinal data. Categorical data can only assume a finite set of distinct values, such as enumerations . In addition, `Pandas` can explicitly specify a column as type categorical when you read the contents of a `CSV` file via the `read_csv()` method.

Interval Data

Interval data pertains to data that is ordered and lies in an interval or range, such as the integers and floating point numbers in the interval `[-1,1]`. Examples of interval data include temperature and income-versus-debt. As you can see, interval data values can be negative as well as positive.

Ratio Data

Ratio data involves measured intervals, such as barometric pressure, height, and altitude. Notice the difference between interval data and ratio data: unlike interval data, ratio data *cannot* be negative. It makes no sense to refer to negative barometric pressure, a person’s height, or altitude above the surface of the earth.

Continuous Data versus Discrete Data

Continuous data can take on any value in an interval, such as `[-1,1]`, `[0,1]`, or `[5,10]`. Hence, continuous data involves floating point numbers, which includes interval data. Keep in mind that an interval contains an uncountably infinite number of values.

One other point to note pertains to possible values and their floating point representation. For instance, a random number in the interval `[0,1]` involves an uncountably infinite number of values, whereas its representation as a floating point number is limited to a large yet finite number of values. Let’s suppose that the integer 10*1000 equals the number of numbers in the interval `[0,1]` that can be represented as a floating point number. Then the smallest positive number in the interval `[0,1]` that can be represented as a floating point number is 1/N. However, there is an uncountably infinite number of values in the interval `[0,1/N)`, which we could approximate as the value 0 (or possibly 1/N).

Discrete data can take on a finite set of values, and the earlier comments regarding successors and predecessors apply to discrete data. As a simple example, the outcome of tossing a coin or throwing a die (or multiple dice) involve discrete data, which are also examples of nominal data. In addition, the associated probabilities for the outcomes form a discrete probability distribution (discussed later).

Qualitative and Quantitative Data

Quantitative data can be either discrete or continuous. For example, a person’s age that is measured in years is discrete, whereas the height of a person is continuous. One point to keep in mind: the word “continuous” in statistics does not always have the same meaning when it is used in a mathematical context. For instance, the price of a house is treated as a continuous feature but it is not continuous in the mathematical sense because the smallest unit of measure is a penny, and there are many (in fact, an uncountably infinite number of) values between two consecutive penny values. Here are two examples of discrete data values, followed by three examples of continuous data values:

- revenue (money)
- number of items sold

- water temperature
- wind speed

- vehicle velocity

Each of the preceding data values are numeric types involving something that has business impact or a physical characteristic.

Qualitative data can sometimes be represented as string-based values, such as different types of color or movie genres. Hence, nominal data and ordinal data are considered qualitative data.

It is possible to use integer-based values for nominal values, such as days of the week and months of the year. In fact, if a dataset contains a string-based feature that is selected as input for a machine learning algorithm, those values are typically converted into integer based values, which can be performed via the `map()` function in `Pandas`. Here are additional examples of qualitative data:

- audio (length)
- pictures or paintings (dimensions)
- text (word count/file size)
- video (length)

Since the items in the preceding list have a parenthetical term that can be used to "measure" the various items, why are they not considered quantifiable and therefore measurable, just like the earlier bullet list? The key difference is that the items in the qualitative items are a form of multimedia, so they do not have a direct and immediate physical characteristic.

However, there are use cases in which media-related data *can* be treated as quantifiable. For example, suppose a company classifies ambient sounds. One practical scenario involves determining if a given sound is a gunshot versus the sound of a backfiring car. As such, the decibel level is an important quantifiable characteristic of both sounds.

In the case of paintings, it is certainly true that they can be “measured” by their selling price, which can sometimes be astronomical.

As another example, consider writers who are paid to write text-based documents. If their payment is based on the number of words in their documents, then the length of a document is a quantifiable characteristic. However, people who read articles typically do not make a distinction between an article that contains 400 words, 450 words, or 500 words.

Finally, the cost of generating a text document that contains the dialogue in a movie can be affected by the length of the movie, in which case videos have a quantifiable characteristic.

Types of Statistical Data

The preceding sections described several data types, whereas this section classifies data types from a statistical standpoint. There are four primary types of statistical data:

- nominal
- ordinal
- interval
- ratio

One way to remember these four types of statistical data is via the acronym `NOIR` (coincidentally the French word for “black”). Please refer to the earlier sections for details regarding any of these data types.

DATA PREPROCESSING

Data preprocessing is the initial stage for validating the contents of a dataset, and it involves a multi-step process (not all the steps are always required):

- removing dependent columns
- handling missing data values
- cleaning text-based data
- removing HTML tags
- removing emoticons/emoticons
- filtering data
- grouping data
- handling currency and date formats

The following subsections contain more information about some of the topics in the preceding bullet list.

Column Dependencies

Another task for data preprocessing pertains to removing dependent columns from a dataset. A *dependent column* is a column that can be derived from one or more columns in a dataset. As a very simple example, if a dataset contains the attributes `length` and `width` that represent the dimensions of rectangular rooms in a house, then the `area` of a room equals the product of the `width` and the `length` of the room. Hence, the area is a feature that depends on the `length` and the `width`, so it is redundant. Alternatively, you might decide to retain the area and the `width` and use their values whenever you need to compute the `length` attribute.

Other important topics pertaining to features in a dataset include collinearity, multicollinearity, VIF (variance inflation factor), homoskedasticity (constant variance of error terms), and heteroskedasticity (non-constant variance of error terms), all of which are discussed later in this book.

Cleaning Datasets

Cleaning datasets involves removing unwanted data, handling missing data, and rectifying erroneous data. In the case of text-based data, you might need to remove HTML tags and punctuation. In the case of data that is supposed to be numeric, it is possible that alphabetic characters are mixed together with numeric data. However, a dataset with numeric features might have incorrect values or missing values (discussed later). In addition, calculating the minimum, maximum, mean, median, and standard deviation of the values of a feature obviously pertain only to numeric values.

After the preprocessing step is completed, data wrangling is performed, which refers to transforming data into a new format, as well as combining features (as new columns and appending rows) from multiple datasets. For example, you might need to convert between different units of measurement (such as date formats and currency types) so that the data values can be represented in a consistent manner in a dataset.

Currency and date values are part of `i18n` (internationalization), whereas `L10n` (localization) targets a specific nationality, language, or region. Hard-coded values (such as text strings) can be stored as resource strings in a file that is often called a *resource bundle*, where each string is referenced via a code. Each language has its own resource bundle.

Now that you have a basic understanding of the nature of various tasks associated with data preprocessing, let’s delve into more details about those tasks, as discussed in the next section.

WORKING WITH DATA TYPES

If you have experience with programming languages, then you know that explicit data types exist (e.g., C, C++, Java, and TypeScript). Some programming languages, such as JavaScript and awk, do not require initializing variables with an explicit type: the type of a variable is inferred dynamically via an implicit type system (i.e., one that is not directly exposed to a developer).

In machine learning, datasets can contain features that have different types of data, such as a combination of one or more of the following types of features:

- numeric data (integer/floating point and discrete/continuous)
- character/categorical data (different languages)
- date-related data (different formats)
- currency data (different formats)
- binary data (yes/no, 0/1, and so forth)
- nominal data (multiple unrelated values)
- ordinal data (multiple and related values)

Consider a dataset that contains real estate data, which can have 30 or more columns, often with the following features:

- the number of bedrooms in a house: numeric value and a discrete value
- the number of square feet: a numeric value and (probably) a continuous value
- the name of the city: character data
- the construction date: a date value
- the selling price: a currency value and probably a continuous value
- the “for sale” status: binary data (either “yes” or “no”)

An example of nominal data is the seasons in a year: although many countries have four distinct seasons, some countries have two distinct seasons. However, keep in mind that seasons can be associated with different temperature ranges (summer versus winter). An example of ordinal data is an employee pay grade: 1=entry level, 2=one year of experience, and so forth. Another example of nominal data is a set of colors, such as {Red, Green, Blue}.

A familiar example of binary data is the pair {Male, Female}, and some datasets contain a feature with these two values. If such a feature is required for training a model, first convert {Male, Female} to a numeric counterpart (such as {0,1}).

A Pandas-based example is here:

```
df['gender'] = df['gender'].map({'Male': 0,
'Female': 1})
```

Similarly, if you need to include a feature whose values are the previous set of colors, you can replace {Red, Green, Blue} with the values {0,1,2}.

WHAT IS DRIFT?

In machine learning terms, *drift* refers to any type of change in distribution over a period of time. *Model drift* refers to a change (drift) in the accuracy of a model's prediction, whereas *data drift* refers to a change in the type of data that is collected. Note that data drift is also called *input drift*, *feature drift*, or *covariate drift*.

There are several factors that influence the value of data, such as accuracy, relevance, and age. For example, physical stores that sell mobile phones are much more likely to sell recent phone models than older models. In some cases, data drift occurs over a period of time, and in other cases, it is because some data is no longer relevant due to feature-related changes in an application. There might be multiple factors that can influence data drift in a specific dataset.

Two techniques for handling data drift are the *domain classifier* and *black-box shift detector*, both of which are discussed online:

https://blog.dataiku.com/towards-reliable-mlops-with-drift-detectors

In addition to the preceding types of drift, other types of changes can occur in a dataset, some of which are listed here:

https://arxiv.org/abs/1912.08142

- concept shift
- covariate shift
- domain shift
- prior probability shift
- spurious correlation shift
- subpopulation shift
- time shift

Perform an online search to find more information about the topics in the preceding list of bullet items. Finally, the following list contains links to open source `Python`-based tools that provide drift detection:

- alibi-detect (*https://github.com/SeldonIO/alibi-detect*)
- evidently (*https://github.com/evidentlyai/evidently*)
- Torchdrift (*http://torchdrift.org/*)

WHAT IS DATA LEAKAGE?

Data leakage occurs when data that is external to the training dataset is included during the training of a model (i.e., inward leaks instead of outward leaks of data). This external data can influence the capability of the model in unexpected ways, which in turn can adversely affect previous metrics that are associated with the model.

Data leakage tends to be an issue with complex datasets, and some of those types are listed here:

- Time series datasets
- Graph problems

- Data stored in multiple files

Two good techniques to minimize data leakage when developing predictive models are as follows:

- Perform data preparation within your cross validation folds.
- Hold back a validation dataset for final sanity check of your developed models.

Generally, it is good practice to use both of these techniques:

https://insidebigdata.com/2014/11/26/ask-data-scientist-data-leakage/

Data Leakage and Differential Privacy

Machine learning models can involve large amounts of data, some of which can be sensitive and confidential. For example, salaries, social security numbers, and medical conditions of people are three obvious examples of confidential information that must not be revealed by a trained model. See the following sources for more information:

- *https://pair.withgoogle.com/explorables/data-leak/?linkId=8028464*
- *https://www.w3schools.in/cyber-security/data-leak-prevention/*
- *https://machinelearningmastery.com/data-leakage-machine-learning/*

Differential privacy prevents training data from leaking confidential information by restricting the information that is available in each data point. Moreover, a model's privacy can be improved through techniques such as aggregation.

MODEL SELECTION AND PREPARING DATASETS

If you have the good fortune to inherit a dataset that is in pristine condition, then data cleaning tasks are vastly simplified: in fact, it might not be necessary to perform *any* data cleaning for the dataset. However, if you need to create a dataset that combines data from multiple datasets that contain different formats for dates and currency, then you need to perform a conversion to a common format. Now let's list some of the criteria for model section, as discussed in the next section.

Model Selection

Model selection refers to the steps involved in selecting a machine learning algorithm. Although there is no "silver bullet" with respect to model selection, here are some simple guidelines:

- Choose a model based on its expected performance.
- Prefer simpler models rather than complex models.
- Use a pre-trained model.

Expected performance refers to the highest accuracy or lowest prediction error, and simpler models generally make fewer assumptions. If you are fortunate, you might find a pre-trained model that enables you to perform some additional training with your custom dataset. Despite these recommendations, always be prepared to try different algorithms and to change the values of some of the hyper parameters of those algorithms.

If you need to train a model that includes features that have categorical data, then you need to convert that categorical data to numeric data. For instance, the Titanic dataset contains a feature called "gender," which is either male or female. As you will see later in this chapter, Pandas makes it extremely simple to "map" male to 0 and female to 1.

Discrete Data versus Continuous Data

As a simple rule of thumb: *discrete* data involves a set of values that can be counted whereas continuous data must be measured. Discrete data can reasonably fit in a drop-down list of values, but there is no exact value for making such a determination. One person might think that a list of 500 values is discrete, whereas another person might think it is continuous.

For example, the list of provinces of Canada and the list of states of the USA are discrete data values, but is the same true for the number of countries in the world (roughly 200) or for the number of languages in the world (more than 7,000)?

Values for temperature, humidity, and barometric pressure are considered *continuous* data types. Currency is also treated as continuous, even though there is a measurable difference between two consecutive values. The smallest unit of currency for US currency is one penny, which is 1/100 of a dollar (accounting-based measurements use the “mil,” which is 1/1,000 of a dollar).

Continuous data types can have subtle differences. For example, someone who is 200 centimeters tall is twice as tall as someone who is 100 centimeters tall, and this is true for a person who is 100 kilograms versus one who is 50 kilograms. However, temperature is different: 80 degrees Fahrenheit is not twice as hot as 40 degrees Fahrenheit.

Furthermore, keep in mind that the word “continuous” has a different meaning in mathematics is not necessarily the same as continuous in machine learning. In the former, a continuous variable (let’s say in the 2D Euclidean plane) can have an uncountably infinite number of values. However, a feature in a dataset that can have more values that can be reasonably displayed in a drop-down list is treated *as though* it is a continuous variable.

For instance, values for stock prices are discrete: they must differ by at least a penny (or some other minimal unit of currency), which is to say, it is meaningless to say that the stock price changes by one-millionth of a penny. However, since there is a plethora of possible stock values, it is treated as a continuous variable. The same comments apply to car mileage, ambient temperature, and barometric pressure.

“Binning” Data Values

The concept of *binning* refers to subdividing a set of values into multiple intervals, and then treating all the numbers in the same interval as though they had the same value. In addition, there are at least three techniques for binning data:

- bins of equal widths
- bins of equal frequency
- bins based on k-means

See the following webpage for more information:

https://towardsdatascience.com/from-numerical-to-categorical-3252cf805ea2

As a simple example of bins of equal widths, suppose that a feature in a dataset contains the age of people in a dataset. The range of values is approximately between 0 and 120, and we could “bin” them into 12 equal intervals, where each consists of 10 values: 0 through 9, 10 through 19, 20 through 29, and so forth.

As another example, using quartiles is even more coarse-grained than the earlier age-related binning example. The issue with binning pertains to the unintended consequences of classifying people in different bins, even though they are in close to each other. For instance, some people struggle financially because they earn a meager wage, and they are also disqualified from financial assistance because their salary is higher than the cut-off point for receiving any assistance.

Scikit-learn provides the `KBinsKDiscretizer` class that uses a clustering algorithm for binning data:

https://scikit-learn.org/stable/modules/generated/scikit-learn.preprocessing.KBinsDiscretizer.html

In case you are interested, a highly technical paper (PDF) with information about clustering and binning is available online:

https://www.stat.cmu.edu/tr/tr870/tr870.pdf

Programmatic Binning Techniques

Earlier in this chapter, you saw a `Pandas`-based example of generating a histogram using data from a `Titanic` dataset. The number of bins was chosen on an ad hoc basis, with no relation to the data itself. However, there are several techniques that enable you to programmatically determine the optimal number of bins, some of which are shown here:

- Doane's formula
- Freedman–Diaconis' Choice
- Rice's Rule
- Scott's Normal Reference Rule
- Square-Root Choice
- Sturge's rule

Doane's formula for calculating the number of bins depends on the number of observations n and the kurtosis (discussed in Chapter 4) of the data, and it is reproduced here:

```
1 + log(n) + log(1 + kurtosis(data) * sqrt(n /
6.0))
```

Freedman–Diaconis' Choice specifies the number of bins for a sample `x`, and it is based on the IQR (InterQuartile Range) and the number of observations `n`, as shown in the following formula:

```
k = 2 * IRQ(x)/[cube root of n]
```

Sturge’s rule to determine the number of bins `k` for Gaussian-based data is based on the number of observations `n`, and it is expressed as follows:

```
k = 1 + 3.322 * log n
```

In addition, after specifying the number of bins `k`, set the minimum bin width `mbw` as follows:

```
mbw = (Max Observed Value – Min Observed Value) /
k
```

Experiment with the preceding formulas to determine which one provides the best visual display for your data. For more information about calculating the optimal number of bins, perform an online search for blog posts and articles.

Potential Issues When Binning Data Values

Partitioning the values of people’s ages as described in the preceding section can be problematic. In particular, suppose that person A, person B, and person C are 29, 30, and 39, respectively. Then person A and person B are probably much more similar to each other than person B and person C, but because of the way in which the ages are partitioned, B is classified as closer to C than to A. In fact, binning can increase Type I errors (false positive) and Type II errors (false negative), as discussed in this blog post (along with some alternatives to binning):

https://medium.com/@peterflom/why-binning-continuous-data-is-almost-always-a-mistake-ad0b3a1d141f

TYPES OF DEPENDENCIES AMONG FEATURES

The ideal case for a dataset involves clean data and fully independent features, which is often not the case in real world datasets. As such, it is important to determine whether there are dependencies among features. This section contains information about the following terms that pertain to feature dependence:

- homoskedasticity
- heteroskedasticity
- collinearity
- multicollinearity
- VIF (variance inflation factor)
- correlation

As a quick summary: *collinearity* adversely affects the significance of independent variables, which in turn reduces the effectiveness of a regression model.

Furthermore, statistical tests such as the student's t-test and `ANOVA` assume homoskedasticity for the data. In other words, such algorithms assume that samples from a population (or even different populations) have the same variance.

Homoskedasticity and Heteroskedasticity

Homoskedasticity refers to the variance of the residuals (actual value minus expected value) in regression models. Recall that one of the assumptions of linear regression is that the residuals have constant homoskedasticity.

Heteroskedasticity is the extent to which features are independent of each other. However, heteroskedasticity is more than just "the opposite" of homoskedasticity, and there are various statistical tests that determine heteroskedasticity, many of which are listed online at

https://en.wikipedia.org/wiki/Heteroskedasticity

Heteroskedasticity can adversely affect statistical tests that assume that errors in a model have the same variance (such as linear regression). The main side effect of heteroskedasticity is its effect on the precision of the estimated values for coefficients.

Homoskedasticity and Linear Regression

Homoskedasticity is one of the assumptions regarding linear regression, as shown in the following list:

- Linearity: The relationship between X and the mean of Y is linear.
- Homoskedasticity: The variance of the residual is the same for any value of X.
- Independence: Observations are independent of each other.
- Normality: For any fixed value of X, Y is normally distributed.

Collinearity

Collinearity refers to a significant level of correlation between two or more predictor variables. Some simple examples are as follows:

- the number of bedrooms and bathrooms in houses
- the height and weight of a person
- education level and income level

Collinearity in machine learning differs from the concept of collinearity in mathematics. For example, three points in the 2D Euclidean plane are called collinear if all three points lie on the same line segment.

Variance Inflation Factor (VIF)

The variance inflation factor (VIF) provides a mechanism for determining the extent to which there is collinearity between two variables. Minimal collinearity is present for VIF values of 1 or 2; values between 5 and 15 indicate moderate collinearity; and values greater than 20 indicate the presence of extreme collinearity.

Multicollinearity

Multicollinearity occurs when the inclusion of two predictor variables in a model *lowers* its statistical significance. VIF values can help you detect the presence of multicollinearity: values greater than 10 indicate a high degree of multicollinearity.

One caveat regarding multicollinearity: two variables can have low correlation and yet contribute to multicollinearity that can only be detected only through the model itself. Also keep in mind that numerous statistical tests, such as the Student's t-test and ANOVA, assume homoskedasticity for the data. In other words, such algorithms assume that samples from a population (or even different populations) have the same variance.

Correlation

Correlation refers to the extent to which a pair of variables are related, which is a number between –1 and 1 inclusive. The most significant correlation values are –1, 0, and 1.

A correlation of 1 means that both variables increase and decrease in the same direction. A correlation of –1 means that both variables increase and decrease in the opposite direction. A correlation of 0 means that the variables are independent of each other.

`Pandas` provides the `corr()` method, which generates a matrix containing the correlation between any pair of features in a data frame. Note that the diagonal values of this matrix are related to the variance of the features in the data frame.

A *correlation matrix* can be derived from a covariance matrix: each entry in the former matrix is a covariance value divided by the product of the standard deviation of the two features in the row and column of a particular entry.

This concludes the portion of the chapter pertaining to dependencies among features in a dataset. The next section discusses different types of currencies that can appear in a dataset, along with a `Python` code sample for currency conversion.

What is a Good Correlation Value?

Although there is no exact value that determines whether a correlation is weak, moderate, or strong, there are some guidelines, as shown here:

- between 0.0 and 0.2: weak
- between 0.2 and 0.5: moderate
- between 0.5 and 0.7: moderately strong
- between 0.7 and 1.0: strong

The preceding ranges are for positive correlations, and the corresponding values for negative correlations are shown here:

- between –0.2 and 0: weak
- between –0.5 and –0.2: moderate
- between –0.7 and –0.5: moderately strong
- between –0.7 and –1.0: strong

However, treat the values in the preceding lists as guidelines: some people classify values between 0.0 and 0.4 as weak correlations, and values between 0.8 and 1.0 as strong correlations. In addition, a correlation of 0.0 means that there is no correlation at all (extra weak?).

Discrimination Threshold

Logistic regression (discussed in Chapter 6) is based on the sigmoid function (which in turn involves Euler's constant) whereby any real number is mapped to a number in the interval (0,1). Consequently, logistic regression is well-suited for classifying binary class membership: i.e., data points that belong to one of two classes. For datasets that contain two class values, let's call them 0 and 1, logistic regression provides a probability that a data point belongs to class 0 or class 1, where the range of probability values includes all the numbers in the interval [0,1].

The *discrimination threshold* is the value whereby larger probabilities are associated with class 1 and smaller probabilities are associated with class 0. Some datasets have a discrimination threshold of 0.5, but in general, this value can be much closer to 0 or 1. Relevant examples include health-related datasets (healthy versus cancer), sports events (win versus lose), and even the DMV (department of motor vehicles), where the latter require 85% accuracy to pass the written test in some US states.

DATA CLEANING AND IMPUTATION

In general, data cleaning involves one or more of the following tasks, which are specific to each dataset:

- Count missing data values.
- Remove/drop redundant columns.
- Assign values to missing data.

- Remove duplicate values.
- Check for incorrect values.
- Ensure uniformity of data.

The following subsections briefly discuss some of the items in the preceding bullet list, along with some Python-based code samples.

Counting Missing Data Values

Listing 1.1 displays the content of `missing_values2.py` that illustrates how to count the missing data values in a `Pandas` data frame.

Listing 1.1: missing_values2.py

```
import pandas as pd
import numpy as np
"""
Count NaN values in one column:
df['column name'].isna().sum()

Count NaN values in an entire data frame:
df.isna().sum().sum()

Count NaN values in one row:
df.loc[[index value]].isna().sum().sum()
"""

data = {'column1':
       [100,250,300,450,500,np.nan,650,700,np.nan],
       'column2' : [
       'X','Y',np.nan,np.nan,'Z','A','B',np.
      nan,np.nan],
        'column3' : [
        'XX',np.nan,'YY','ZZ',np.nan,np.nan, 'AA'
        ,np.nan,np.nan]
        }
df = pd.DataFrame(data,columns=
['column1','column2', 'column3'])

print("dataframe:")
print(df)

print("Missing values in 'column1':")
print(df['column1'].isna().sum())

print("Total number of missing values:")
print(df.isna().sum().sum())

print("Number of missing values for row index 7 (=
row #8):")
print(df.loc[[7]].isna().sum().sum())
```

Listing 1.1 starts with two `import` statements and a comment block that explains the purpose of several `Pandas` methods pertaining sums of values and the `isna()` method for finding `NaN` values in a dataset.

The next portion of Listing 1.1 initializes a dictionary with three arrays of values that are used to initialize the `Pandas` data frame `df`. Next, the missing values in `column1` are displayed, followed by the number of missing values in every column of `df`. The final code block displays the number of missing values for the row whose index is 7. Launch the code in Listing 1.1 and you will see the following output:

```
dataframe:
    column1 column2 column3
```

```
0     100.0       X      XX
1     250.0       Y     NaN
2     300.0     NaN      YY
3     450.0     NaN      ZZ
4     500.0       Z     NaN
5       NaN       A     NaN
6     650.0       B      AA
7     700.0     NaN     NaN
8       NaN     NaN     NaN
Missing values in 'column1':
2
Total number of missing values:
11
Number of missing values for row index 7 (= row
#8):
2
```

Navigate to this link where you find additional `Python` code samples for data cleaning:

https://lvngd.com/blog/data-cleaning-with-python-pandas/

Drop Redundant Columns

Listing 1.2 displays the content of `drop_columns.py` that illustrates how to remove redundant columns from a Pandas data frame.

Listing 1.2: drop_columns.py

```
import pandas as pd

# specify a valid CSV file here:
```

```
df1 = pd.read_csv("my_csv_file.csv") # <= specify
your own CSV file

# remove redundant columns:
df2 = df1.drop(['url'],axis=1)

# remove columns with over 50% missing values
df3 = df2.dropna(thresh=half_count,axis=1)
```

Listing 1.2 initializes the `Pandas` data frame `df1` with the contents of the `CSV` file `my_csv_file.csv` and then initializes the `Pandas` data frame `df2` with the contents of `df1`, and then drops the column `url`, or some other column that exists in your `CSV` file. Finally, the `Pandas` data frame `df3` is initialized with the contents of `Pandas` data frame `df2`, after which columns are dropped if they have more than 50% missing values.

Remove Duplicate Data Values

Data deduplication refers to the task of removing row-level duplicate data values. Refer to Chapter 3, which contains a Python code sample that shows you how to find and remove duplicate data values from a dataset. You can also read about data deduplication online at

https://www.data4v.com/python-dedupe-library-machine-learning-to-de-duplicate-data/

Uniformity of Data Values

An example of uniformity of data involves verifying that the data in a given feature contains the same units measure. For example, the following set of values have numeric values that are in a narrow range, but the units of measure are incorrect (note the absence of a space between the numbers and the units of measure):

```
50mph
50kph
100mph
20kph
```

Listing 1.3 displays the content of `same_units.sh` that illustrates how to ensure that items in a set of strings have the same unit of measure.

Listing 1.3: same_units.sh

```
strings="120kph 100mph 50kph"
new_unit="fps"

for x in `echo $strings`
do
  number=`echo $x | tr -d [a-z][A-Z]`
  unit=`echo $x | tr -d [0-9]`
  echo "initial: $x"
  new_num="${number}${new_unit}"
  echo "new_num: $new_num"
  echo
done
```

Listing 1.3 starts by initializing the variables `strings` and `new_unit`, followed by a `for` loop that iterates through each string in the `strings` variable. During each iteration, the variables `number` and `unit` are initialized with the characters and digits, respectively, in the current string represented by the loop variable `x`. Next, the variable `new_num` is initialized as the concatenation of the contents of `number` and `new_unit`. Launch the code in Listing 1.3 and you will see the following output:

```
initial: 120kph
new_num: 120fps

initial: 100mph
new_num: 100fps

initial: 50kph
new_num: 50fps
```

Too Many Missing Data Values

Datasets with mostly N/A values, which is to say, 80% or more are N/A or NaN values, is always daunting, but not necessarily hopeless. As a simple first step, you can drop rows that contain N/A values, which might result in a loss of 99% of the data. A variation of the preceding all-or-nothing step for handling datasets with a majority of N/A values is as follows:

- Use a kNN imputer to fill missing values in high value columns.
- Drop low priority columns that have > 50% missing values.
- Use a KNN imputer (again) to fill the remaining missing values.
- Try using 3 or 5 as the # of nearest neighbors.

The preceding sequence attempts to prune insignificant data to concentrate on reconstructing the higher priority columns through data imputation. Of course, there is no guaranteed methodology for salvaging such a dataset, so you need some ingenuity as you experiment with datasets containing highly limited data values. If the dataset is highly imbalanced, consider *oversampling* before you drop columns and/or rows, which is discussed o chapter 2.

Categorical Data

Categorical values are usually discrete and can easily be encoded by specifying a number for each category. If a category has `n` distinct values, then visualize the `nxn` identity matrix: each row represents one of the distinct values.

This technique is called *one-hot encoding*, and you can use the `OneHotEncoder` class in `scikit-learn` by specifying the dataset `X` and also the column index to perform one-hot encoding:

```
from scikit-learn.preprocessing import
OneHotEncoder
ohc = OneHotEncoder(categorical_features = [0])
X = onehotencoder.fit_transform(X).toarray()
```

Since each one-hot encoded row contains one 1 and (n–1) zero values, one-hot encoding is an inefficient technique. Another technique involves the Pandas `map()` function that replaces string values with a single column that contains numeric values. For example, the following code block replaces `Male` and `Female` with 0 and 1, respectively:

```
values = {'Male' : 0, 'Female' : 1}
df['gender'] = df['gender'].map(values)
```

A variation of the preceding is the following code block:

```
data['gender'].replace(0, 'Female',inplace=True)
data['gender'].replace(1, 'Male',inplace=True)
```

Another variation of the preceding code is this code block:

```
data['gender'].replace([0,1],
['Male','Female'],inplace=True)
```

The `Pandas map()` function converts invalid entries to `NaN`.

Data Inconsistency

Data inconsistency occurs when distinct value are supposed to be the same value, such as "smith" and "SMITH" instead of "Smith." Another example would be "YES," "Yes," "YS," and "ys" instead of "yes." In all cases except for "ys," you can convert all the other strings to lower case, which replaces all the strings with "smith" or "yes," respectively.

Mean Value Imputation

Listing 1.4 displays the content of `mean_imputation.py` that shows you how to replace missing values with the mean value of each feature.

Listing 1.4: mean_imputation.py

```
import numpy as np
import pandas as pd
import random

filename="titanic.csv"
df = pd.read_csv(filename)

# display null values:
print("=> Initial df.isnull().sum():")
print(df.isnull().sum())
print()

# replace missing ages with mean value:
df['age'] = df['age'].fillna(df['age'].mean())

"""
```

```
Or use median(), min(), or max():
df['age'] = df['age'].fillna(df['age'].median())
df['age'] = df['age'].fillna(df['age'].min())
df['age'] = df['age'].fillna(df['age'].max())
"""

# FILL MISSING DECK VALUES WITH THE mode():
mode = df['deck'].mode()[0]
#df['deck'] = df['deck'].fillna(mode)

print("=> new age and deck values:")
print([df[['deck','age']]])
```

Listing 1.4 starts with several `import` statements and then initializes the variable `filename` with the name of the Titanic dataset. The next portion of code initializes the variable `df` with the contents of the `CSV` file and then displays the number of rows, for each feature, that contain null values.

The next portion of Listing 1.4 replaces missing values in the `age` feature with the mean value of the non-null values. Notice the comment block that shows you how to specify the median, minimum, or maximum value in the `age` feature. Next, the variable `mode` is initialized with the mode of the deck feature. The final code snippet displays the updated values for the deck and the age features. Launch the code in Listing 1.4 and you will see the following output:

```
=> Initial df.isnull().sum():
survived       0
pclass         0
sex            0
age          177
sibsp          0
```

```
parch            0
fare             0
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
alone            0
dtype: int64

=> new age and deck values:
[   deck         age
0      C  22.000000
1      C  38.000000
2      C  26.000000
3      C  35.000000
4      C  35.000000
..   ...        ...
886    C  27.000000
887    B  19.000000
888    C  29.699118
889    C  26.000000
890    C  32.000000

[891 rows x 2 columns]]
```

Random Value Imputation

Random value imputation involves generating random values and using those values to replace missing values in a dataset. Listing 1.5 displays the content of `random_imputation.py` that shows you how to replace missing values with random values that are selected from within a given feature.

Listing 1.5: random_imputation.py

```
import numpy as np
import pandas as pd
import random

filename="titanic.csv"
df = pd.read_csv(filename)

# display null values:
print("=> Initial df.isnull().sum():")
print(df.isnull().sum())
print()

# replace missing ages with mean value:
df['age'] = df['age'].fillna(df['age'].mean())

#Randomize missing column data
def randomize_deck(df2):
  df = df2.copy()
  data = df["deck"]
  mask = data.isnull()
  samples = random.choices(data[~mask].values, k =
  mask.sum())
  data[mask] = samples
  return df

# FILL MISSING DECK VALUES WITH RANDOM non-null
values:
```

```
df = randomize_deck(df)

print("=> new age and deck values:")
print([df[['deck','age']]])
```

Listing 1.5 starts with several import statements and then initializes the variable `filename` with the name of the Titanic dataset. The next portion of code initializes the variable `df` with the contents of the `CSV` file and then displays the number of rows, for each feature, that contain null values.

The next portion of Listing 1.5 replaces missing values in the `age` feature with the mean value of the age feature. Next, the Python function `randomize_deck()` randomizes missing values in the deck feature. The final portion of Listing 1.5 invokes the `randomize_deck()` function and then displays the new values for the deck feature and the age feature. Launch the code in Listing 1.5 and you will see the following output:

```
=> Initial df.isnull().sum():
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
```

```
alone         0
dtype: int64

=> new age and deck values:
[    deck        age
0       D  22.000000
1       C  38.000000
2       C  26.000000
3       C  35.000000
4       B  35.000000
..    ...        ...
886     E  27.000000
887     B  19.000000
888     D  29.699118
889     C  26.000000
890     E  32.000000

[891 rows x 2 columns]]
```

Multiple Imputation

Rather than replacing each missing value in a dataset with one randomly imputed value, it may make sense to replace each with several imputed values that reflect our uncertainty about our imputation model. For example, if we impute using a regression model, we may want our imputations to reflect not only sampling variability (as random imputation should), but also our uncertainty about the regression coefficients in the model. If these coefficients themselves are modeled, we can draw a new set of missing-value imputations, for each draw from the distribution of the coefficients.

Multiple imputation does this by creating several (say, five) imputed values for each missing value, each of which is predicted from a slightly different model and each of which also reflects sampling variability. How do we analyze these data? The simple idea is to use each set of imputed values to form (along with the observed data) a completed dataset. Within each completed dataset, a standard analysis can be run. Then inferences can be combined across datasets.

Matching and Hot-Deck Imputation

Hot-deck imputation is performed by randomly selecting another row that has similar values on other variables and use its value in the row that contains missing values.

You can also impute values by using an existing value that appears in a similar data point, which can also be used recommendation systems that utilizes user-user collaboration to impute ratings for movies.

For example, suppose that data collected for a dataset involves estimating a risk factor for each data point, where the risk is derived information from an associated document. However, the information might be insufficient to calculate an accurate risk value. One potential solution involves finding the nearest neighbors to the new data point and then calculating the average of the risk values in the nearest neighbors.

Is a Zero Value Valid or Invalid?

In general, replace a missing numeric value with zero is a risky choice: this value is obviously incorrect if the values of a feature are positive numbers between 1,000 and 5,000 (or some other range of positive numbers). For a feature that has numeric values, replacing a missing value with the mean of existing values can be better than the value zero (unless the average equals zero); also consider using the median value. For categorical data, consider using the mode to replace a missing value.

There are situations where you can use the mean of existing values to impute missing values, but not the value zero, and vice versa. As a first example, suppose that an attribute contains the height in centimeters of a set of persons. In this case, the mean could be a reasonable imputation, whereas 0 suffers from the following:

1. It is an invalid value (nobody has height 0).
2. It will skew statistical quantities such as the mean and variance.

You might be tempted to use the mean instead of 0 when the minimum allowable value is a positive number, but use caution when working with highly imbalanced datasets. As a second example, consider a small community of 50 residents with

1. 45 people have an average annual income of USD 50,000
2. 4 other residents have an annual income of 10,000,000
3. 1 resident has an unknown annual income

Although the preceding example might seem contrived, it is likely that the median income is preferable to the mean income, and certainly better than imputing a value of 0.

As a third example, suppose that a company generates weekly sales reports for multiple office branches, and a new office has been opened, but has yet to make any sales. In this case, the use of the mean to impute missing values for this branch would produce fictitious results. Hence, it makes sense to use the value 0 for all sales-related quantities in the new branch office, which will accurately reflect the sales-related status of the new branch.

Data Normalization

Normalization is the process of scaling numeric columns in a dataset so that they have a common scale. In addition, the scaling is performed as follows:

1. scaling values to the range [0,1]
2. without losing information
3. without distorting any differences that exist in the ranges of values.

You can perform data normalization via the function `MinMaxScaler()` in the `scikit-learn` library.

Assigning Classes to Data

Listing 1.6 displays the content of `product_prices.csv` and Listing 1.7 displays the content of `assign_classes.py` that illustrates how to assign a class value to each row in a dataset.

Listing 1.6: product_prices.csv

```
item,price
product1,100
product2,200
product3,250
product4,300
product5,400
```

Listing 1.7: assign_classes.py

```
import pandas as pd

df = pd.read_csv("product_prices.csv")
print("contents of df:")
print(df)
print()
```

```
# define class ranges:
def class_value2(y):
  if y<=100:
    return '(1) 0 - 100'
  elif y<=200:
    return '(2) 100 - 200'
  elif y<=250:
    return '(3) 200 - 250'
  else:
    return '(4) 250+'

def class_value(y):
  if y<=100:
    return '1'
  elif y<=200:
    return '2'
  elif y<=250:
    return '3'
  else:
    return '4'

df['class1'] = df['price'].apply(class_value)
df['class2'] = df['price'].apply(class_value2)

print("contents of df:")
print(df)
```

Listing 1.7 initializes the `Pandas` data frame `df` with the contents of the `CSV` file `product_prices.csv` (displayed in Listing 1.6) and displays its contents. The next portion of Listing 1.7 is the `Python` function `class_value2`, which returns a string whose contents are a range of values that are based on the parameter `y`. For example, if `y` is at most 100, the function returns the string `(1)  0 - 100`, and similar strings for larger values of `y`.

The next portion of Listing 1.7 is the `Python` function `class_value`, which returns a string 1, 2, 3, or 4, depending on the parameter `y`. The last portion of Listing 1.7 initializes the column `class1` and `class2` in `df` by invoking the `apply()` method with the `Python` functions `class_value` and `class_value2`, respectively. Launch the code in Listing 1.7 and you will see the following output:

```
contents of df:
      item price
0  product1   100
1  product2   200
2  product3   250
3  product4   300
4  product5   400

contents of df:

       item  price class1         class2
0  product1    100      1  (1)    0 - 100
1  product2    200      2  (2) 100 - 200
2  product3    250      3  (3) 200 - 250
3  product4    300      4         (4) 250+
4  product5    400      4         (4) 250+
```

Other Data Cleaning Tasks

As a quick review, here are additional tasks that belong to data cleaning that might be relevant to a given dataset:

- Detect outliers/anomalies.
- Resolve missing data.
- Resolve incorrect data.
- Resolve duplicate data.
- Remove hidden control characters (ex: \t, ^L, and ^M).
- Remove HTML tags (ex: <div> and <a>).
- Handle diacritical marks.
- Check for gaps in sequences of data.
- Check for unusual distributions.
- Examine the actual data instead of relying on documentation.

The appendix contains `Python`-based code samples that use regular expressions for cleaning data.

DeepChecks and Data Validation

`DeepChecks` is a `Python` module that enables you to specify a set of rules to validate data in a dataset, and its home page is

https://deepchecks.com/

A `DeepChecks` *suite* contains one or more *check*s, where a check displays pass/fail output, depending on the outcome of the check. Moreover, *conditions* can be added, modified, or removed from a check, and similarly for each check in a suite. If you have written `Java` unit tests, then `DeepChecks` might remind you of `Junit,` which was written by Kent Beck. the creator of the “Extreme Programming” methodology.

SUMMARY

This chapter started with an explanation of datasets, a description of data wrangling, and details regarding various types of data. Then you learned about techniques for scaling numeric data, such as normalization and standardization. You saw how to convert categorical data to numeric values, and how to handle dates and currency.

In addition, you learned about the notion of data drift and data leakage, followed by model selection. You also learned about how to map categorical data to numeric data.

Furthermore, you learned about concepts such as homoskedasticity, collinearity, variance inflation factor, and correlation. Finally, you saw Python-based code samples that involve currency and date values, and along with an example of assigning class values to data.

OceanofPDF.com

CHAPTER 2

OUTLIER AND ANOMALY DETECTION

This chapter shows you how to process outliers, anomalies, and missing data, as well as data cleaning and data wrangling techniques. In addition, this chapter includes short Python code samples that use NumPy as well as Pandas to find outliers, how to calculate z-scores, and how to count the number of missing values in a dataset.

The first part of this chapter discusses the relationship among fraud, anomalies, and outliers, along with Python code samples that illustrate how to find outliers. The second section discusses fraud detection (there are many types), along with anomaly detection. You will also learn about algorithms such as SMOTE for handling imbalanced classes in a dataset. The third section contains details regarding the bias-variance tradeoff and various types of statistical bias.

IMPORT STATEMENTS FOR THIS CHAPTER

This chapter contains a mixture of Python-based code samples, an awk-based shell script, and a Java code sample to show you how to solve tasks using different technologies. All the code samples are straightforward, and if you can follow the Pandas and awk-based code samples in Chapter 1, then you will most likely be able to understand the code samples in this chapter.

This chapter requires basic knowledge of Python and Pandas, such as creating Pandas data frames, as well as reading and writing CSV files. Knowledge of the awk programming language is required for three shell scripts that invoke the `awk` command if you decide to read those code samples.

In addition, the following list contains all the `import` statements that you will encounter in the Python code samples for this chapter:

```
from forex_python.bitcoin import BtcConverter
from forex_python.converter import CurrencyRates
from imblearn.over_sampling import SMOTE
from scipy import stats
from sklearn.covariance import EllipticEnvelope
from sklearn.linear_model import
LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import
train_test_split
import numpy as np
import pandas as pd
import sys
```

WORKING WITH OUTLIERS

In brief, an *outlier* is an abnormal data value that is outside the range of "normal" values in a dataset. For example, a person's height in centimeters is typically between 30 centimeters and 250 centimeters, which means that a height of 5 centimeters or a height of 500 centimeters is an outlier because those values are not possible for humans.

Outliers in a dataset are significantly larger or smaller than the inliers in a dataset. Outliers exist for various reasons, such as data variability, experimental errors, or erroneous measurements. In addition, outliers can create issues during statistical analysis, such as adversely affecting the value of the mean and the standard deviation. Three types of outliers are explained at

https://pub.towardsai.net/the-7-stages-of-preparing-data-for-machine-learning-dfe454da960b

Outlier Detection/Removal

There are techniques available that help you detect outliers in a dataset, as shown in the following bullet list, along with a one-line description and links for additional information:

- IQR
- z-score
- trimming
- winsorizing
- minimum covariance determinant
- local outlier factor
- Huber and Ridge
- isolation forest (tree-based algorithm)
- one-class SVM

The IQR (interquantile range) algorithm detects data points that are outside of 1.5 times of an interquartile range that either lie above the 3rd quartile or lie below the 1st quartile. Such points can be considered outliers.

The *z-score* for data points involves subtracting the mean and then dividing by the standard deviation:

```
Z-score = (X-mean)/std
```

In general, z-scores that are greater than 3 are considered outliers, but you can adjust this value (e.g., 2.5 or 2) that is more suitable for your dataset.

Perhaps *trimming* is the simplest technique (apart from dropping outliers), which involves removing rows whose feature value is in the upper 5% range or the lower 5% range. *Winsorizing* the data is an improvement over trimming: set the values in the top 5% range equal to the maximum value in the 95th percentile, and set the values in the bottom 5% range equal to the minimum in the 5th percentile.

The *Minimum Covariance Determinant* is a covariance-based technique, and a Python-based code sample that uses this technique is downloadable:

https://scikit-learn.org/stable/modules/outlier_detection.html

Two other techniques involve the `Huber` and the `Ridge` classes, both of which are included as part of Scikit-learn. The `Huber` error is less sensitive to outliers because it is calculated via linear loss, similar to the MAE (Mean Absolute Error). A code sample that compares `Huber` and `Ridge` is downloadable from

https://scikit-learn.org/stable/auto_examples/linear_model/plot_huber_vs_ridge.html

You can also explore the Theil-Sen estimator and RANSAC, which are “robust” against outliers:

https://scikit-learn.org/stable/auto_examples/linear_model/plot_theilsen.html

https://en.wikipedia.org/wiki/Random_sample_consensus

Four algorithms for outlier detection are discussed online at

https://www.kdnuggets.com/2018/12/four-techniques-outlier-detection.html

One other scenario involves “local” outliers. For example, suppose that you use k-means (or some other clustering algorithm) and determine that a value is an outlier with respect to one of the clusters. While this value is not necessarily an “absolute” outlier, detecting such a value might be important for your use case.

Incorrectly Scaled Values versus Outliers

You already know that an outlier is a value that is significantly different from the other values for a given feature. Now suppose that a numeric feature has a set of values in the range [90,100], but the correct range of values for this feature is [9,10]. Notice that the incorrect values do not contain any outliers, and also that those values can easily be scaled the range [0,1] using a technique described in chapter 1.

However, suppose that you are not a domain expert for the data in this dataset, so you do not realize that the initial values are out of range for that feature. As a result, you proceed to scale these data values so that they are in the range [0,1]. Although it is possible to train a model with this scaled dataset, the newly scaled values (as well as the initial values) are incorrect. Unfortunately, errors can arise when you perform other operations with this data, such as calculating the correlation between this feature and some other feature in the dataset. Hence, it is important to have domain knowledge to detect and rectify this type of error.

Other Outlier Techniques

If you want to explore additional techniques for detecting outliers, the following techniques are also available:

- Modified Z-score
- MAD (Median Absolute Deviation)
- Tukey's boxplot
- Carling Median Rule

In brief, the modified z-score provides a more fine-trained set of values that sometimes detect outliers that are not detected via a standard z-score. The MAD technique uses a median-based technique (instead of mean and variance values) that is less sensitive to outliers, which is better suited for non-normal distributions. Tukey's boxplot uses quartile values, whereas the Carling Median Rule uses median values. Perform an online search for more details, formulas, and code samples regarding the outlier techniques in the preceding bullet list.

Outliers and XGBoost

XGBoost is a tree-based ensemble classification algorithm, along with the random forest multi-tree classification algorithm, both of which are well-known in machine learning. XGBoost *automatically* handles outliers in a dataset, whereas the random forest algorithm does *not* automatically handle outliers.

Of course, there are several other differences between the XGBoost algorithm and the random forest algorithm (such as early stopping and the learning rate) that might influence which algorithm you select to train a model using your dataset. If this scenario is relevant to your use case, navigate to the following URLs for more information regarding XGBoost and random forest:

https://scikit-learn.org/stable/modules/generated/scikit-learn.ensemble.RandomForestClassifier.html

https://www.datacamp.com/community/tutorials/xgboost-in-python

FINDING OUTLIERS WITH NUMPY

We have not discussed the `NumPy` library in depth. We will only use the `NumPy` `array()`, `mean()`, and `std()` methods in this section, all of which have intuitive functionality.

Listing 2.1 displays the content of `numpy_outliers1.py` that illustrates how to use `NumPy` methods to find outliers in an array of numbers.

Listing 2.1: numpy_outliers1.py

```
import numpy as np

arr1 = np.array([2,5,7,9,9,40])
print("values:",arr1)

data_mean = np.mean(arr1)
data_std = np.std(arr1)
print("data_mean:",data_mean)
print("data_std:" ,data_std)
print()

multiplier = 1.5
cut_off = data_std * multiplier
lower = data_mean - cut_off
upper = data_mean + cut_off
```

```
print("lower cutoff:",lower)
print("upper cutoff:",upper)
print()

outliers = [x for x in arr1 if x < lower or x >
upper]
print('Identified outliers: %d' % len(outliers))
print("outliers:",outliers)
```

Listing 2.1 starts by defining a `NumPy` array of numbers and then calculates the mean and standard deviation of those numbers. The next block of code initializes two numbers that represent the upper and lower values that are based on the value of the `cut_off` variable. Any numbers in the array `arr1` that lie to the left of the lower value or to the right of the upper value are treated as outliers.

The final section of code in Listing 2.1 initializes the variable-`outliers` with the numbers that are determined to be outliers, and those values are printed. Launch the code in Listing 2.1 and you will see the following output:

```
values: [ 2 5 7 9 9 40]
data_mean: 12.0
data_std: 12.754084313139327

lower cutoff: -7.131126469708988
upper cutoff: 31.1311264697089

Identified outliers: 1
outliers: [40]
```

The preceding code sample specifies a hard-coded value to calculate the upper and lower range values.

Listing 2.2 is an improvement in that you can specify a set of values from which to calculate the upper and lower range values, and the new block of code is shown in bold.

Listing 2.2: numpy_outliers2.py

```
import numpy as np

arr1 = np.array([2,5,7,9,9,40])
print("values:",arr1)

data_mean = np.mean(arr1)
data_std = np.std(arr1)
print("data_mean:",data_mean)
print("data_std:" ,data_std)
print()

multipliers = np.array([0.5,1.0,1.5,2.0,2.5,3.0])
for multiplier in multipliers:
   cut_off = data_std * multiplier
   lower, upper = data_mean - cut_off, data_mean + cut_off
   print("=> multiplier: ",multiplier)
   print("lower cutoff:",lower)
   print("upper cutoff:",upper)

   outliers = [x for x in df['data'] if x < lower or x > upper]
   print('Identified outliers: %d' % len(outliers))
   print("outliers:",outliers)
   print()
```

Listing 2.2 contains a block of new code that initializes the variable `multipliers` as an array of numeric values that are used for finding outliers. Although you will probably use a value of 2.0 or larger on a real dataset, this range of numbers can give you a better sense of detecting outliers. Launch the code in Listing 2.2 and you will see the following output:

```
values: [ 2 5 7 9 9 40]
data_mean: 12.0
data_std: 12.754084313139327

lower cutoff: -7.131126469708988
upper cutoff: 31.1311264697089

Identified outliers: 1
outliers: [40]
=> multiplier:  0.5
lower cutoff: 5.622957843430337
upper cutoff: 18.377042156569665
Identified outliers: 3
outliers: [2, 5, 40]

=> multiplier:  1.0
lower cutoff: -0.7540843131393267
upper cutoff: 24.754084313139327
Identified outliers: 1
outliers: [40]

=> multiplier:  1.5
lower cutoff: -7.131126469708988
upper cutoff: 31.1311264697089
Identified outliers: 1
```

```
outliers: [40]

=> multiplier:  2.0
lower cutoff: -13.508168626278653
upper cutoff: 37.50816862627865
Identified outliers: 1
outliers: [40]

=> multiplier:  2.5
lower cutoff: -19.88521078284832
upper cutoff: 43.88521078284832
Identified outliers: 0
outliers: []

=> multiplier:  3.0
lower cutoff: -26.262252939417976
upper cutoff: 50.26225293941798
Identified outliers: 0
outliers: []
```

FINDING OUTLIERS WITH PANDAS

The `Pandas` code sample in this section involves a very simple `Pandas` data frame, the `mean()` method, and the `std()` method.

Listing 2.3 displays the content of `pandas_outliers1.py` that illustrates how to use `Pandas` to find outliers in an array of numbers.

Listing 2.3: pandas_outliers1.py

```
import pandas as pd
```

```
df = pd.DataFrame([2,5,7,9,9,40])
df.columns = ["data"]

print("=> complete data set:")
print(df)
print()

data_mean = df['data'].mean()
data_std = df['data'].std()
print("=> data_mean:",data_mean)
print("=> data_std: ",data_std)
print()

multiplier = 1.5
cut_off = data_std * multiplier
lower, upper = data_mean - cut_off, data_mean +
cut_off
print("=> lower cutoff:",lower)
print("=> upper cutoff:",upper)
print()

# outliers: method #1
outliers = [x for x in df['data'] if x < lower or
x > upper]
print('=> Identified outliers: %d' %
len(outliers))
print("=> outliers (#1):",outliers)
print()

# outliers: method #2
outliers = [x for x in df['data'] if x < lower or
x > upper]
```

```
outliers = df[(df.data < lower) | (df.data >
upper)]
print('=> Identified outliers: %d' %
len(outliers))
print("=> outliers (#2):",outliers)
print()

# keep the inliers and drop the outliers:
df = df[(df.data > lower) & (df.data < upper)]
print("=> inliers without outliers:")
print(df)
print()
```

Listing 2.3 starts by defining a `Pandas` data frame and then calculates the mean and standard deviation of those numbers. The next block of code initializes two numbers that represent the upper and lower values that are based on the value of the `cut_off` variable. Any numbers in the data frame that lie to the left of the lower value or to the right of the upper value are treated as outliers.

The final section of code in Listing 2.3 initializes the variable `outliers` with the numbers that are determined to be outliers by means of a `Python` comprehension, and those values are printed, whereas the second technique accomplishes the same result without a `Python` comprehension. Launch the code in Listing 2.3 and you will see the following output:

```
=> complete data set:
        data
0          2
1          5
```

```
2          7
3          9
4          9
5         40

=> data_mean: 12.0
=> data_std: 13.971399357258385

=> lower cutoff: -8.957099035887577
=> upper cutoff: 32.95709903588758

=> Identified outliers: 1
=> outliers (#1): [40]

=> Identified outliers: 1
=> outliers (#2): data
5         40

=> inliers without outliers:
        data
0          2
1          5
2          7
3          9
4          9
```

The preceding code sample specifies a hard-coded value to calculate the upper and lower range values.

Listing 2.4 is an improvement over Listing 2.3 in that you can specify a set of values from which to calculate the upper and lower range values, and the new block of code is shown in bold.

Listing 2.4: pandas_outliers2.py

```
import pandas as pd

#df = pd.DataFrame([2,5,7,9,9,40])
#df = pd.DataFrame([2,5,7,8,42,44])

df = pd.DataFrame([2,5,7,8,42,492])
df.columns = ["data"]
print("=> data values:")
print(df['data'])

data_mean = df['data'].mean()
data_std = df['data'].std()
print("=> data_mean:",data_mean)
print("=> data_std:" ,data_std)
print()

multipliers = [0.5,1.0,1.5,2.0,2.5,3.0]
  for multiplier in multipliers:
  cut_off = data_std * multiplier
  lower, upper = data_mean - cut_off, data_mean +
  cut_off
  print("=> multiplier: ",multiplier)
  print("lower cutoff:",lower)
  print("upper cutoff:",upper)

  outliers = [x for x in df['data'] if x < lower or
  x > upper]
  print('Identified outliers: %d' % len(outliers))
  print("outliers:",outliers)
  print()
```

Listing 2.4 contains a block of new code that initializes the variable `multipliers` as an array of numeric values that are used for finding outliers. Although you will probably use a value of 2.0 or larger on a real dataset, this range of numbers can give you a better sense of detecting outliers. Launch the code in Listing 2.4 and you will see the following output:

```
=> data values:
0            2
1            5
2            7
3            8
4           42
5          492
Name: data, dtype: int64
=> data_mean: 92.66666666666667
=> data_std: 196.187325448579

=> multiplier: 0.5
lower cutoff: -5.42699605762283
upper cutoff: 190.76032939095617
Identified outliers: 1
outliers: [492]

=> multiplier: 1.0
lower cutoff: -103.52065878191233
upper cutoff: 288.85399211524566
Identified outliers: 1
outliers: [492]

=> multiplier: 1.5
lower cutoff: -201.614321506218
```

```
upper cutoff: 386.9476548395352
Identified outliers: 1
outliers: [492]

=> multiplier: 2.0
lower cutoff: -299.7079842304913
upper cutoff: 485.0413175638247
Identified outliers: 1
outliers: [492]

=> multiplier: 2.5
lower cutoff: -397.80164695478084
upper cutoff: 583.1349802881142
Identified outliers: 0
outliers: []

=> multiplier: 3.0
lower cutoff: -495.8953096790703
upper cutoff: 681.2286430124036
Identified outliers: 0
outliers: []
```

Calculating Z-scores to Find Outliers

The `z-score` of a set of numbers is calculated by standardizing those numbers, which involves 1) subtracting their mean from each number, and 2) dividing by their standard deviation. Although you can perform these steps manually, the `Python SciPy` library simplifies the steps involved. If need be, you can install this package with the following command:

```
pip3 install scipy
```

Listing 2.5 displays the content of `outliers_zscores.py` that illustrates how to find outliers in an array of numbers. As you will see, this code sample relies on convenience methods from `NumPy`, `Pandas`, and `SciPy`.

Listing 2.5: outliers_zscores.py

```
import numpy as np
import pandas as pd
from scipy import stats

arr1 = np.array([2,5,7,9,9,40])
print("values:",arr1)

df = pd.DataFrame(arr1)

zscores = np.abs(stats.zscore(df))
print("z scores:")
print(zscores)
print()

upper = 2.0
lower = 0.5
print("=> upper outliers:")
print(zscores[np.where(zscores > upper)])
print()

print("=> lower outliers:")
print(zscores[np.where(zscores < lower)])
print()
```

Listing 2.5 starts with several `import` statements, followed by initializing the variable `arr1` as a `NumPy` array of numbers, and then displaying the values in `arr1`. The next code snippet initializes the variable `df` as a data frame that contains the values in the variable `arr1`.

Next, the variable `zscores` is initialized with the z-scores of the elements of the `df` data frame, as shown here:

```
zscores = np.abs(stats.zscore(df))
```

The next section of code initializes the variables `upper` and `lower`, and the z-scores whose values are less than the value of `lower` or greater than the value `upper` are treated as outliers, and those values are displayed. Launch the code in Listing 2.5 and you will see the following output:

```
values: [ 2  5  7  9  9 40]
z scores:
[[0.78406256]
 [0.54884379]
 [0.39203128]
 [0.23521877]
 [0.23521877]
 [2.19537517]]

=> upper outliers:
[2.19537517]

=> lower outliers:
[0.39203128 0.23521877 0.23521877]
```

FINDING OUTLIERS WITH SCIKIT-LEARN (OPTIONAL)

This section is optional because the code involves the `EllipticEnvelope` class in `scikit-learn.covariance`, which we do not cover in this book. However, it is still worthwhile to peruse the code and compare this code with earlier code samples for finding outliers.

Listing 2.6 displays the content of `elliptic_envelope_outliers.py` that illustrates how to use `Pandas` to find outliers in an array of numbers.

Listing 2.6: elliptic_envelope_outliers.py

```
# pip3 install scikit-learn
from scikit-learn.covariance import
EllipticEnvelope
import numpy as np

# Create a sample normal distribution:
Xdata = np.random.normal(loc=5, scale=2,
size=10).reshape(-1, 1)
print("Xdata values:")
print(Xdata)
print()

# instantiate and fit the estimator:
envelope = EllipticEnvelope(random_state=0)
envelope.fit(Xdata)

# create a test set:
```

```
test = np.array([0, 2, 4, 6, 8, 10, 15, 20, 25,
30]).reshape(-1, 1)
print("test values:")
print(test)
print()

# predict() returns 1 for inliers and -1 for
outliers:
print("envelope.predict(test):")
print(envelope.predict(test))
```

Listing 2.6 starts with several `import` statements and then initializes the variable `Xdata` as a column vector of random numbers from a Gaussian distribution. The next code snippet initializes the variable envelope as an instance of the `EllipticEnvelope` from `scikit-learn` (which will determine if there are any outliers in `Xdata`), and then trained on the data values in `Xdata`.

The next portion of Listing 2.6 initializes the variable `test` as a column vector, much like the initialization of `Xdata`. The final portion of Listing 2.6 makes a prediction on the values in the variable test and also displays the results: the value -1 indicates an outlet. Launch the code in Listing 2.6 and you will see the following output:

```
Xdata values:
[[5.21730452]
 [5.49182377]
 [2.87553776]
 [4.20297013]
 [8.29562026]
 [5.78097977]
 [4.86631006]
 [5.47184212]
 [4.77954946]
 [8.66184028]]

test values:
[[ 0]
 [ 2]
 [ 4]
 [ 6]
 [ 8]
 [10]
 [15]
 [20]
 [25]
 [30]]

envelope.predict(test):
[-1 1 1 1 1 -1 -1 -1 -1 -1]
```

FRAUD DETECTION

According to one estimate (Crowe Global, “Fraud Costs”) , worldwide fraud amounts to more than five trillion dollars.

Earlier sections in this chapter discussed how outliers differ from inliers in terms of their value, frequency, or location, or some combination. Inliers are common occurrences, which is to say, there is nothing unusual about the values for inliers. An outlier draws attention to the possibility of fraud but does not necessarily indicate that fraud has occurred.

An *anomaly* is also an outlier that is a more serious type of outlier: there is a greater chance that this type of outlier is also fraud. By way of analogy, consider a traffic light consisting of green (go), yellow (caution), and red (stop). An outlier can be in any of the following ranges:

- between green and red
- between green and yellow
- between yellow and red

As such, there are different levels of caution involved with anomalies and outliers. Specifically, an anomaly belongs to the third category, whereas a “benign” outlier (which is *not* an anomaly) is in the second category. Moreover, the collection of all types of outliers is in the first category. With the preceding points in mind, here is a short list of various types of fraud:

- credit card fraud
- payroll fraud
- insurance fraud

Although there is no single method for always determining fraud, there are some techniques for detecting potentially fraudulent transactions. For example, if you encounter a suspicious event for a customer, calculate the following values for that customer:

- total purchase amount for this day
- number of transactions for this day
- time of day for each transaction
- number of locations
- addresses of those locations

Now compare the values in the preceding list with the customer daily transaction patterns to see if there is a likely case of fraud. In case you are interested, the following webpage contains a list of 41 types of fraud, along with techniques for fraud prevention:

https://www.i-sight.com/resources/41-types-of-fraud-and-how-to-detect-and-prevent-them/

TECHNIQUES FOR ANOMALY DETECTION

First, let's keep in mind that an anomaly is also an outlier: the difference is that the consequences of an anomaly can be much worse than an outlier. For example, consider credit card purchases whereby a person who living in San Francisco suddenly makes credit card purchases in New York City. A one-time purchase could be an anomaly (i.e., a stolen credit card), or it could be a purchase made during a short stop-over en route to a vacation in another city or country (i.e., a Type I error). A business trip or a vacation in New York City would probably involve a larger set of credit card purchases, and therefore comprise normal purchases instead of credit card theft.

Consider a variation of the preceding scenario: a customer on a business trip in New York City has his credit card stolen and then multiple credit card purchases are made in San Francisco. The latter might escape detection because the customer lives in San Francisco (i.e., a Type II error). However, if multiple credit card purchases are made simultaneously in San Francisco and New York City during the same period of time, there is a greater risk of anomalous behavior because a spouse making credit card purchases with a card that is linked to the same bank account would have a different credit card number.

Incidentally, credit card companies *do* provide a telephone menu option to "notify us of any upcoming business or travel plans," which can help reduce the possibility of Type I or Type II errors associated with credit card purchases.

In addition to credit card fraud, there are many other types of fraud, such as insurance fraud and or payroll fraud.

Before we explore this topic, it is worth noting that various types of machine learning algorithms are available for detecting anomalies. One type involves classification algorithms, such as kNN, decision trees, and SVMs. Another type involves unsupervised algorithms, such as autoencoders (a deep learning architecture), GMM (Gaussian Mixture Models), kMeans (a well-known clustering algorithm), and PCA.

However, since this book is not primarily about machine learning or deep learning algorithms, this chapter discusses other techniques for anomaly detection. Note that kNN is discussed later in this chapter in the section regarding imputation of missing data values, and decision trees are relevant to entropy and the Gini impurity.

One other technique for anomaly detection uses a Bayesian network, which is a probabilistic graphical model (PGM). Bayesian networks and PGMs are outside the scope of this book, but the following webpage contains information about anomaly detection using a Bayesian network:

https://www.bayesserver.com/docs/techniques/anomaly detection

Selecting an Anomaly Detection Technique

Unfortunately, there is no simple way to *decide* how to deal with anomalies and outliers in a dataset. Although you can drop rows that contain outliers, doing so might deprive the dataset (and therefore the trained model) of valuable information. You can try modifying the data values, but again, this might lead to erroneous inferences in the trained model.

Another possibility is to train a model with the dataset that contains anomalies and outliers, and then train a model with a dataset from which the anomalies and outliers have been removed. Compare the two results and see if you can infer anything meaningful regarding the anomalies and outliers. In addition, various techniques are available for anomaly detection, some of which are listed here:

- LOF
- HBOS
- PyOD
- Numeric Outlier (IQR)
- Z-Score
- DBSCAN
- Isolation Forest

The *Local Outlier Factor* (LOF) technique is an unsupervised technique that calculates a local anomaly score via the kNN (k Nearest Neighbor) algorithm. Documentation and short code samples that use LOF are available online:

https://scikit-learn.org/stable/modules/generated/scikit-learn.neighbors.LocalOutlierFactor.html

https://towardsdatascience.com/outlier-detection-theory-visualizations-and-code-a4fd39de540c

LOF: Local Outlier Factor

Local Outlier Factor (LOF) is included in this portion of the chapter because LOF is a density-based algorithm for anomaly detection. LOF is used in an unsupervised setting to find out local anomalies in the data. Typically, global anomalies can be easily found by other techniques. However, local anomalies are not detected via other algorithms because they appear in groups. This is precisely why the density-based technique is preferred over distance-based technique.

HBOS

HBOS is an acronym for Anomaly Detection with Histogram-based Outlier Score. HBOS starts with the construction of a histogram for a variable. The height of the bin that contains the data point can be used into the outlier score. Since we prefer a small outlier score for inliers data and a large score for outliers, we can invert the height of a bin to be used as the outlier score of the data point of a variable.

The maximum height of each histogram is normalized to 1.0. This ensures all the univariate scores can be summed up with equal weight. The following formula gives the specific description: Assuming there are d variables (dimensions) and p data points. The HBOS is the sum of the logarithmic outlier score of all features.

PyOD

PyOD is an acronym for Python Outlier Detection (PyOD), which is an open source Python-based library for anomaly detection. PyOD collects a wide range of techniques ranging from supervised learning to unsupervised learning techniques. Some techniques work better than others for a given dataset. PyOD includes at least 20 anomaly detection techniques such as PCA, kNN, AutoEncoder, SOS, and XGB.

https://medium.com/dataman-in-ai/anomaly detection-with-histogram-based-outlier-detection-hbo-bc10ef52f23f

For the series on PyOD, please read the following articles:

- Anomaly Detection with Histogram-based Outlier Score: The Histogram-based Outlier Score
- Anomaly Detection with PyOD: the k Nearest Neighbors (KNN)
- Anomaly Detection with Autoencoders Made Easy: Autoencoders
- Use the Isolated Forest with PyOD: Isolated Forest

- *https://lilianweng.github.io/lil-log/2021/12/05/semi-supervised-learning.html*

Numeric Outlier (IQR)

This is the simplest nonparametric outlier detection method in a one-dimensional feature space. Here, outliers are calculated by means of the IQR (InterQuartile Range). The first and the third quartile (Q1, Q3) are calculated. An outlier is then a data point `xi` that lies outside the interquartile range. Using the interquartile multiplier value k=1.5, the range limits are the typical upper and lower whiskers of a box plot.

Z-Score

The z-score is a parametric outlier detection method in a one- or low-dimensional feature space. This technique assumes a Gaussian distribution of the data. The outliers are the data points that are in the tails of the distribution and therefore far from the mean. How far depends on a set threshold `zthr` for the normalized data points `zi` that are based on the xi values, and calculated with the formula:

```
zi = (xi-mu)/sigma (where mu = mean)
```

An outlier is then a normalized data point which has an absolute value greater than `zthr`.

DBSCAN

In high level terms, this technique is based on the DBSCAN clustering method. DBSCAN is a nonparametric, density-based outlier detection method in a one- or multi-dimensional feature space. In the DBSCAN clustering technique, all data points are defined either as Core Points, Border Points, or Noise Points.

Core Points are data points that have at least `MinPts` neighboring data points within a distance ε. Border Points are neighbors of a Core Point within the distance ε, but with less than `MinPts` neighbors within the distance ε. All other data points are Noise Points that are identified as outliers. Outlier detection thus depends on the required number of neighbors `MinPts`, the distance ε, and the selected distance measure, like Euclidean or Manhattan.

Isolation Forest

This is a nonparametric method for large datasets in a one- or multi--dimensional feature space. An important concept in this method is the isolation number. The *isolation number* is the number of splits needed to isolate a data point. This number of splits is ascertained by following these steps:

- A point "a" to isolate is selected randomly.
- A random data point "b" is selected that is between the minimum and maximum value and different from "a."
- If the value of "b" is lower than the value of "a," the value of "b" becomes the new lower limit.
- If the value of "b" is greater than the value of "a," the value of "b" becomes the new upper limit.
- This procedure is repeated as long as there are data points other than "a" between the upper and the lower limit.

It requires fewer splits to isolate an outlier than it does to isolate a non-outlier; i.e., an outlier has a lower isolation number in comparison to a non-outlier point. A data point is therefore defined as an outlier if its isolation number is lower than the threshold.

The threshold is defined based on the estimated percentage of outliers in the data, which is the starting point of this outlier detection algorithm:

https://towardsdatascience.com/are-these-data-normal-anomalies-outliers-in-machine-learning-a259bbe58690

An explanation with images of the isolation forest technique is available at

https://quantdare.com/isolation-forest-algorithm

The following Python code block contains an example of using the `IsolationForest` class that is available in scikit-learn.

```
from scikit-learn.ensemble import IsolationForest
import pandas as pd
clf = IsolationForest(max_samples=100,
random_state=42)
table = pd.concat([input_table['Mean(ArrDelay)']],
axis=1)
clf.fit(table)
output_table = pd.DataFrame(clf.predict(table))
```

Deep Learning and Anomaly Detection

Although deep learning is outside the scope of this book, it is still worthwhile to know something about this topic. Deep learning architectures can be used for anomaly detection, such as RNNs, LSTMs, GANs, and transformers.

This webpage discusses LSTMs for time series anomaly detection:

https://www.renom.jp/notebooks/tutorial/time_series/lstm-anomalydetection/notebook.html

This link discusses RNNs for time series anomaly detection:

https://ieeexplore.ieee.org/document/7486356

If you prefer to delegate the task of anomaly detection, there are various services available, some of which are as follows:

- Anodot
- Outlier.ai
- Vectra Cognito
- QuickSight (Amazon)
- Sherlock (Yahoo)

Perform an online search for articles that discuss deep learning and anomaly detection, as well as the products that are in the preceding bullet list.

WORKING WITH IMBALANCED DATASETS

Imbalanced datasets contain at least once class that has significantly more values than another class in the dataset. For example, if class A has 99% of the data and class B has 1%, which classification algorithm would you use?

Unfortunately, classification algorithms do not work as well with highly imbalanced datasets. However, there are various techniques that you can use to reduce the imbalance in a dataset. Regardless of the technique that you decide to use, keep in mind the following detail: *resampling techniques are only applied to the training data* (not the validation data or the test data).

In addition, if you perform k-fold cross validation on a training set, then oversampling is performed in each fold during the training step. To avoid data leakage, make sure that you do *not* perform oversampling prior to k-fold cross validation.

Data Sampling Techniques

Data sampling techniques reduce the imbalance in imbalanced datasets, and some well-known techniques are as follows:

- random resampling: rebalances the class distribution
- random undersampling: deletes examples from the majority class
- random oversampling: duplicates data in the minority class
- SMOTE (Synthetic Minority Oversampling Technique)

Random resampling rebalances the class distribution by resampling the data space to reduce the discrepancy between the number of rows in the majority class and the minority class.

The *random undersampling* technique removes samples that belong to the majority class from the dataset, and involves the following:

- randomly removes samples from majority class
- can be performed with or without replacement
- alleviates imbalance in the dataset
- may increase the variance of the classifier
- may discard useful or important samples

However, random undersampling does not work well with extremely unbalanced datasets, such as a 99% and 1% split into two classes. Moreover, undersampling can result in losing information that is useful for a model.

Random oversampling generates new samples from a minority class: this technique duplicates examples from the minority class.

Another option to consider is the Python package imbalanced-learn in the scikit-learn-contrib project. This project provides various resampling techniques for datasets that exhibit class imbalance. More details are available online:

https://github.com/scikit-learn-contrib/imbalanced-learn

Another well-known technique is `SMOTE`, which involves data augmentation (i.e., synthesizing new data samples). `SMOTE` was initially developed by means of the `kNN` algorithm (other options are available), and it can be an effective technique for handling imbalanced classes.

Removing Noisy Data

There are several techniques that attempt to remove “noisy data,” which is often near the boundary, so that there is less ambiguity in the classification of the remaining data. Some of these techniques are listed here:

- Near Miss
- Condensed Nearest Neighbor (CNN)
- Tomek links
- ENN (Edited Nearest Neighbor)
- OSS (One-Sided Selection)
- Neighborhood Cleaning Rule (NCR)

Keep in mind that CNN in the bullet list is different from Convolutional Neural Network. In addition, one potential drawback to CNN is due to its random choice of sample points. Tomek links is an undersampling technique that modifies CNN in two ways. One improvement involves finding pairs of data points (x,y) that are cross-class nearest neighbors, which is to say, x and y belong to different classes and x and y also have the smallest Euclidean distance. After finding all such pairs, the values that belong to the majority class are removed.

However, the efficacy of Tomek links does vary, and it is often used in conjunction with other undersampling techniques (including CNN). The following code snippet for Tomek links shows you how to import the appropriate class from `imblearn`:

```
from imblearn.under_sampling import TomekLinks
// details omitted
undersampled = TomekLinks()
```

ENN (Edited Nearest Neighbor) removes data points in the majority class that are misclassified as belonging to the minority class. ENN uses a “pairing” technique to find a matching nearest neighbor in the majority class that is paired with a “noisy” or ambiguous point that is located along the class boundary, and then removes the point in the majority.

The following code snippet for ENN links shows you how to import the appropriate class from `imblearn`:

```
from imblearn.under_sampling import
EditedNearestNeighbours
// details omitted
undersampled =
EditedNearestNeighbours(n_neighbors=5)
```

The default value for `n_neighbors` is 3, whereas the value 5 is specified in the preceding code snippet.

Cost-Sensitive Learning

Before you read this section, keep in mind that it involves the confusion matrix. If you are unfamiliar with this matrix, please read the relevant material in Chapter 4 before proceeding with the material in this section.

Cost-sensitive learning refers to assigning different costs for the misclassification of data points, which is often relevant to imbalanced datasets. For example, the consequences of a Type II error (false negative) are considerably worse than a Type I error (false positive) for datasets pertaining to fraud detection, medical diagnosis (such as cancer), and so forth.

Specifically, the *cost* in cost-sensitive learning refers to the penalty that is assigned to an incorrect prediction. As such, the goal is to minimize the overall cost of a given model during the training step.

Chapter 4 discusses the *confusion matrix* for a binary classification, which has four possibilities:

- true positive
- false positive
- true negative
- false negative

An example of a confusion matrix is shown here, followed by the interpretation of the values in the confusion matrix:

```
[[60  4]
 [16 20]]
```

The four values in the preceding 2x2 matrix represent the following quantities:

```
True positive: 60

False positive: 4
True negative: 20
False negative: 16
```

Thus, the main diagonal consists of correct predictions whereas the “off” diagonal consists of incorrect predictions. Cost-sensitive learning involves defining a cost matrix from the values in the confusion matrix. An example of a cost matrix is shown here:

```
[[0 5]
  [50 0]]
```

The value 50 is much larger than the value 5 because 50 is the cost associated with a false negative, whereas 5 is associated with a false positive, and zero cost is associated with correct predictions.

In this example, the associated cost function that we wish to minimize is shown here:

```
Cost = 50*FN + 5*FP
```

Detecting Imbalanced Data

This step involves counting the number of rows that are associated with each class. First, read the dataset into a Pandas data frame (let's call it `df`) and then invoke the following code snippet:

```
df['your-target-column'].value_counts()
```

An example of the output from the preceding code snippet is here:

```
[OUT]
1          21000
0           6000
```

As you can see in the output above, class 1 is more than 3 times larger than class 0, so this column in the dataset is imbalanced.

The Python-based open source library scikit-learn a vast set of algorithms for machine learning, some of which support a `class_weight` parameter, as listed here:

- logistic regression
- perceptron
- random forest
- SVM

Simply set the value of `class_weight` parameter equal to `balanced` before training the chosen model. Incidentally, it might also be worthwhile to train the chosen model with the default value for the `class_weight` parameter.

Rebalancing Datasets

Let's return to the example of a dataset for which class `A` has 99% of the data and class `B` has 1%. Which classification algorithm would you use? The following list contains several well-known techniques for handling imbalanced datasets (not in any particular order):

- random resampling (rebalances the class distribution)
- random oversampling (duplicates data in the minority class)
- random undersampling (deletes examples from the majority class)
- algorithm selection
- cross validation for imbalanced data
- generating synthetic data (ex: SMOTE)
- performance metric selection

Any of the preceding techniques can be utilized for imbalanced datasets, and keep in mind that your results may vary, so it's worth trying more than one technique.

Specify Stratify in Data Splits

This step is straightforward because Scikit-Learn supports a `stratify` parameter that ensures the data is split so that the training data and test data contain the same proportion of class values. For example, if a dataset contains 60% and 40%, respectively, of class `A` and class `B` in a column, then the train data and test data will contain the same proportions for class `A` and `B`.

The following code block demonstrates how to split a dataset where `X` and `y` have already been initialized so that the data is stratified:

```
from scikit-learn.model_selection import
train_test_split

X_train,X_test,y_train,y_test =
                         train_test_split(X,
                                          y,
                                          test_size=0.25,
                                          random_state=42,
                                          stratify=y)
```

SUMMARY

This chapter started with a discussion regarding the relationship among fraud, anomalies, and outliers, along with Python code samples that illustrate how to find outliers. The second section discusses fraud detection (there are many types), along with anomaly detection. Next, you learned about algorithms, such as SMOTE, for handling imbalanced classes in a dataset. Finally, you learned about the bias-variance tradeoff and various types of statistical bias.

REFERENCE

Fraud costs the global economy over US$5 trillion (no date) *Crowe Global*. Available at: *http://www.crowe.com/global/news/fraud-costs-the-global-economy-over-us$5-trillion* (Accessed: December 13, 2022).

OceanofPDF.com

CHAPTER 3

CLEANING DATASETS

This chapter shows you how to clean datasets, which includes finding missing data, incorrect data, and duplicate data. In some cases, you might also decide to consolidate data values (e.g., treat the prefix “Mr.,” “MR,” and “mr” as the same label).

The first part of this chapter contains several Pandas code samples that use Pandas to read CSV files and then calculate statistical values such as the mean, median, mode, and standard deviation.

The second part of this chapter uses Pandas to handle missing values in CSV files, starting with CSV files that contain a single column, followed by two-column CSV files. These code samples will prepare you to work with multi-column CSV files, such as the custom `bmi.csv` file and the Titanic `titanic.csv` file.

After you have completed this chapter, you will be ready to learn how to split CSV files into subregions that are then processed via classification algorithms, such as kNN, decision trees, and random forests.

PREREQUISITES FOR THIS CHAPTER

This chapter contains a mixture of Python-based code samples and an awk-based shell script. All the code samples are straightforward, and if you can follow the Pandas and awk-based code samples in Chapter 1, then you will most likely be able to understand the code samples in this chapter.

This chapter requires basic knowledge of Python and Pandas, such as creating Pandas data frames, as well as reading and writing CSV files. Knowledge of the awk programming language is required for three shell scripts that invoke the `awk` command, if you decide to read those code samples.

In addition, the following list contains all the `import` statements that you will encounter in the Python code samples for this chapter:

- `from forex_python.bitcoin import BtcConverter`
- `from forex_python.converter import CurrencyRates`
- `from fuzzywuzzy import fuzz`
- `from fuzzywuzzy import process`
- `from imblearn.over_sampling import SMOTE`
- `from scipy import stats`
- `from sklearn.covariance import EllipticEnvelope`
- `from sklearn.linear_model import LogisticRegression`
- `from sklearn.metrics import classification_report`
- `from sklearn.metrics import confusion_matrix`
- `from sklearn.model_selection import train_test_split`
- `import numpy as np`
- `import pandas as pd`

ANALYZING MISSING DATA

This section contains subsections that describe types of missing data, common causes of missing data, and various ways to inpute values for missing data. Outlier detection, fraud detection, and anomaly detection pertain to analyzing *existing* data.

By contrast, missing data presents a different issue, which in turn raises the following question: what can you do about the missing values? Is it better to discard data points (e.g., rows in a CSV file) with missing values, or is it better to estimate reasonable values as a replacement for the missing values? Missing data can adversely affect a thorough analysis of a dataset, whereas erroneous data can increase bias and uncertainty.

At this point, you have undoubtedly realized that a single solution does not exist for every dataset: you need to perform an analysis on a case-by-case basis after you have learned some of the techniques that might help you effectively address missing data values.

Causes of Missing Data

There are various reasons for missing values in a dataset, some of which are listed here:

- values are unavailable
- values were improperly collected
- inaccurate data entry

Although you might be tempted to *always* replace a missing value with a concrete value, there are situations in which you cannot determine a value. As a simple example, a survey that contains questions for people under 30 will have a missing value for respondents who are over 30, and in this case specifying a value for the missing value would be incorrect. With these details in mind, there are various ways to fill missing values, some of which are listed here:

- Remove rows with a high percentage of missing values (50% or larger).
- Use one-hot encoding for categorical data.
- Use the `Imputer` class from scikit-learn library.
- Fill missing values with the values in an adjacent row.
- Replace missing data with the mean/median/mode value.
- Infer ("impute") the value for missing data.

Once again, the technique that you select for filling missing values is influenced by various factors, such as

- how you want to process the data
- the type of data involved
- the cause of missing values (see above)

Although the most common technique involves the mean value for numeric features, someone needs to determine which technique is appropriate for a given feature.

However, if you are not confident that you can impute a reasonable value, consider deleting the row with a missing value, and then train a model with the imputed value and also with the deleted row.

One problem that can arise after removing rows with missing values is that the resulting dataset is too small. In this case, consider using SMOTE (Synthetic Minority Oversampling Technique), which is discussed later in this chapter, to generate synthetic data.

PANDAS, CSV FILES, AND MISSING DATA

This section contains several subsections with `Python`-based code samples that create `Pandas` data frames and then replace missing values in the data frames. First, we'll look at small `CSV` files with one column and then we'll look at small `CSV` files with two columns. Later, we'll look at skewed `CSV` files as well as multi-row `CSV` files.

Single Column CSV Files

Listing 3.1 displays the content of the `CSV` file `one_char_column1.csv`, and Listing 3.2 displays the content of `one_char_column1.py` that fills in missing values in the `CSV` file.

Listing 3.1: one_char_column1.csv

```
gender
Male
Male
NaN
Female
Male
```

Listing 3.2: one_char_column1.py

```
import pandas as pd

df1 = pd.read_csv('one_char_column1.csv')

print("=> initial dataframe contents:")
print(df1)
print()

df = df1.fillna("FEMALE")
print("dataframe after fillna():")
print(df)
print()
```

Listing 3.2 starts with two `import` statements and then initializes the Pandas data frame `df1` with the contents of `one_char_column1.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace missing values with the string `FEMALE`. Launch the code in Listing 3.2, and you will see the following output:

```
=> initial dataframe contents:
         gender
0          Male
1          Male
2           NaN
3        Female
4          Male

dataframe after fillna():
         gender
0          Male
1          Male
2        FEMALE
3        Female
4          Male
```

Listing 3.3 displays the content of the CSV file `one_char_column2.csv`, and Listing 3.4 displays the content of `one_char_column2.py` that fills in missing values in the CSV file.

Listing 3.3: one_char_column2.csv

```
gender
Male
Male
Null
Female
Male
```

Listing 3.4: one_char_column2.py

```
import pandas as pd

df1 = pd.read_csv('one_char_column1.csv')

print("=> initial dataframe contents:")
print(df1)
print()

df = df1.fillna("FEMALE")
print("dataframe after fillna():")
print(df)
print()
```

Listing 3.4 starts with two `import` statements and then initializes the `Pandas` data frame `df1` with the contents of `one_char_column1.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace missing values with the string `FEMALE`. Launch the code in Listing 3.4 and you will see the following output:

```
=> initial dataframe contents:
         gender
0          Male
```

```
1           Male
2           Null
3         Female
4           Male

df after fillna():
         gender
0           Male
1           Male
2           Null
3         Female
4           Male

gender mode: Male

=> first mapped dataframe:
         gender
0           Male
1           Male
2         Female
3            NaN
4           Male

=> second mapped dataframe:
         gender
0           Male
1           Male
2         Female
3         Female
4           Male
```

Listing 3.5 displays the content of the `CSV` file `one_numeric_column.csv`, and Listing 3.6 displays the content of `one_numeric_column.py` that fills in missing values in the `CSV` file.

Listing 3.5: one_numeric_column.csv

```
age
19
np.nan
16
NaN
17
```

Listing 3.6: one_numeric_column.py

```
import pandas as pd
import numpy as np
df1 = pd.read_csv('one_numeric_column.csv')
df2 = df1.copy()
print("=> initial dataframe contents:")
print(df1)
print()

maxval = 12345
df1['age'] = df1['age'].map({'np.nan' : maxval})
print("=> dataframe after map():")
print(df1)
print()

# refresh contents of df1:
df1 = df2
df1['age'] = df1['age'].fillna(maxval)
```

```
print("=> refreshed dataframe after fillna():")
print(df1)
print()

df1 = df1.fillna(777)
print("dataframe after second fillna():")
print(df1)
print()

#print(df1.describe())
# error due to np.nan value:
#df1['age'].astype(int)

cols = df1.select_dtypes(np.number).columns
df1[cols] = df1[cols].fillna(9876)
print("df1 after third fillna():")
print(df1)
print()

# => this code block works:
#df1 = df1.replace('np.nan', 9876)
df1 = df1.replace({'np.nan': 9876})
print("df1 after replace():")
print(df1)
print()
```

Listing 3.6 starts with two `import` statements and then initializes the `Pandas` data frame `df1` with the contents of `one_numeric_column.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace missing values with the value `9876`. Launch the code in Listing 3.6, and you will see the following output:

```
=> initial dataframe contents:
            age
0            19
1        np.nan
2            16
3           NaN
4            17

=> dataframe after map():
            age
0           NaN
1       12345.0
2           NaN
3           NaN
4           NaN

=> refreshed dataframe after fillna():
            age
0            19
1        np.nan
2            16
3         12345
4            17

dataframe after second fillna():
            age
0            19
1        np.nan
2            16
```

```
3          12345
4             17

df1 after third fillna():
            age
0            19
1        np.nan
2            16
3         12345
4            17

df1 after replace():
            age
0            19
1          9876
2            16
3         12345
4            17
```

Two-Column CSV Files

Listing 3.7 displays the content of the `CSV` file `two_columns.csv`, and Listing 3.8 displays the content of `two_columns.py` that fills in missing values in the `CSV` file.

Listing 3.7: two_columns.csv

```
gender,age
Male,19
Male,np.nan
NaN,16
Female,NaN
```

```
Male,17
```

Listing 3.8: two_columns.py

```
import pandas as pd

df1 = pd.read_csv('two_columns.csv')

print("=> initial dataframe contents:")
print(df1)
print()

df1 = df1.fillna("MISSING")
print("dataframe after fillna():")
print(df1)
print()

df1 = df1.replace({'np.nan': 99})
print("dataframe after replace():")
print(df1)
print()
```

Listing 3.8 starts with two `import` statements and then initializes the `Pandas` data frame `df1` with the contents of `two_columns.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace `NA` values with the string `MISSING`, followed by a code block that replaces `NaN` values with 99. Launch the code in Listing 3.8, and you will see the following output:

```
=> initial dataframe contents:
        gender      age
0         Male       19
1         Male   np.nan
```

```
2             NaN         16
3          Female        NaN
4            Male         17

dataframe after fillna():
         gender        age
0          Male         19
1          Male     np.nan
2       MISSING         16
3        Female    MISSING
4          Male         17

dataframe after replace():
         gender        age
0          Male         19
1          Male         99
2       MISSING         16
3        Female    MISSING
4          Male         17
```

Listing 3.9 displays the content of the `CSV` file `two_columns2.csv`, and Listing 3.10 displays the content of `two_columns2.py` that fills in missing values in the `CSV` file.

Listing 3.9: two_columns2.csv

```
gender,age
Male,19
Male,NaN
NaN,16
Female,18
Male,17
```

Listing 3.10: two_columns2.py

```
import pandas as pd

df1 = pd.read_csv('two_columns2.csv')
df2 = df1.copy()
print("=> initial dataframe contents:")
print(df1)
print()

# calculates the mean value on the
# 'age' column and skips NaN values:
full_avg = df1.mean()
print("full_avg:")
print(full_avg)
print()

avg = df1['age'].mean()
print("average age:",avg)
print()

#WRONG: *all* values are updated:
#df1['age'] = df1['age'].mean()
#print(df1)
#print()

# refresh contents of df1:
df1 = df2

# fillna() replaces NaN with avg:
df1['age'] = df1['age'].fillna(avg)
print("updated age NaN with avg:")
print(df1)
print()

#this does not replace NaN with avg:
#df1 = df1.replace({'NaN': avg})

mode = df1['gender'].mode()[0]
print("mode:",mode)

df1['gender'] = df1['gender'].fillna(mode)
print("updated gender NaN with mode:")
print(df1)
print()
```

Listing 3.10 starts with two `import` statements and then initializes the `Pandas` data frame `df1` with the contents of `two_columns2.csv`, after which its contents are displayed. The next code block initializes the variable `avg` with the mean value of the `age` column. This value is used to update all missing values in the `age` attribute in the data frame `df1`, as shown here:

```
df1['age'] = df1['age'].fillna(avg)
```

The next portion of Listing 3.10 resets the contents of `df1` to its initial contents, followed by a code snippet that updates only the missing values in the `avg` column with the average, as shown here:

```
df1['age'] = df1['age'].fillna(avg)
```

The next section of code initializes the variable `mode` with the mode of the gender column, replaces the missing values in the `gender` column with the value of the variable `mode`, and then prints the updated contents of the data frame `df1`. Launch the code in Listing 3.10, and you will see the following output:

```
=> initial dataframe contents:
         gender      age
0          Male     19.0
1          Male      NaN
2           NaN     16.0
3        Female     18.0
4          Male     17.0

full_avg:
age    17.5
dtype: float64

average age: 17.5

updated all with same avg:
         gender      age
0          Male     17.5
1          Male     17.5
2           NaN     17.5
3        Female     17.5
4          Male     17.5

updated age NaN with avg:
         gender      age
0          Male     19.0
1          Male     17.5
2           NaN     16.0
3        Female     18.0
4          Male     17.0
```

```
mode: Male
updated gender NaN with mode:
         gender      age
0          Male     19.0
1          Male     17.5
2          Male     16.0
3        Female     18.0
4          Male     17.0
```

MISSING DATA AND IMPUTATION

In general, data cleaning involves or more of the following tasks, which are specific to each dataset:

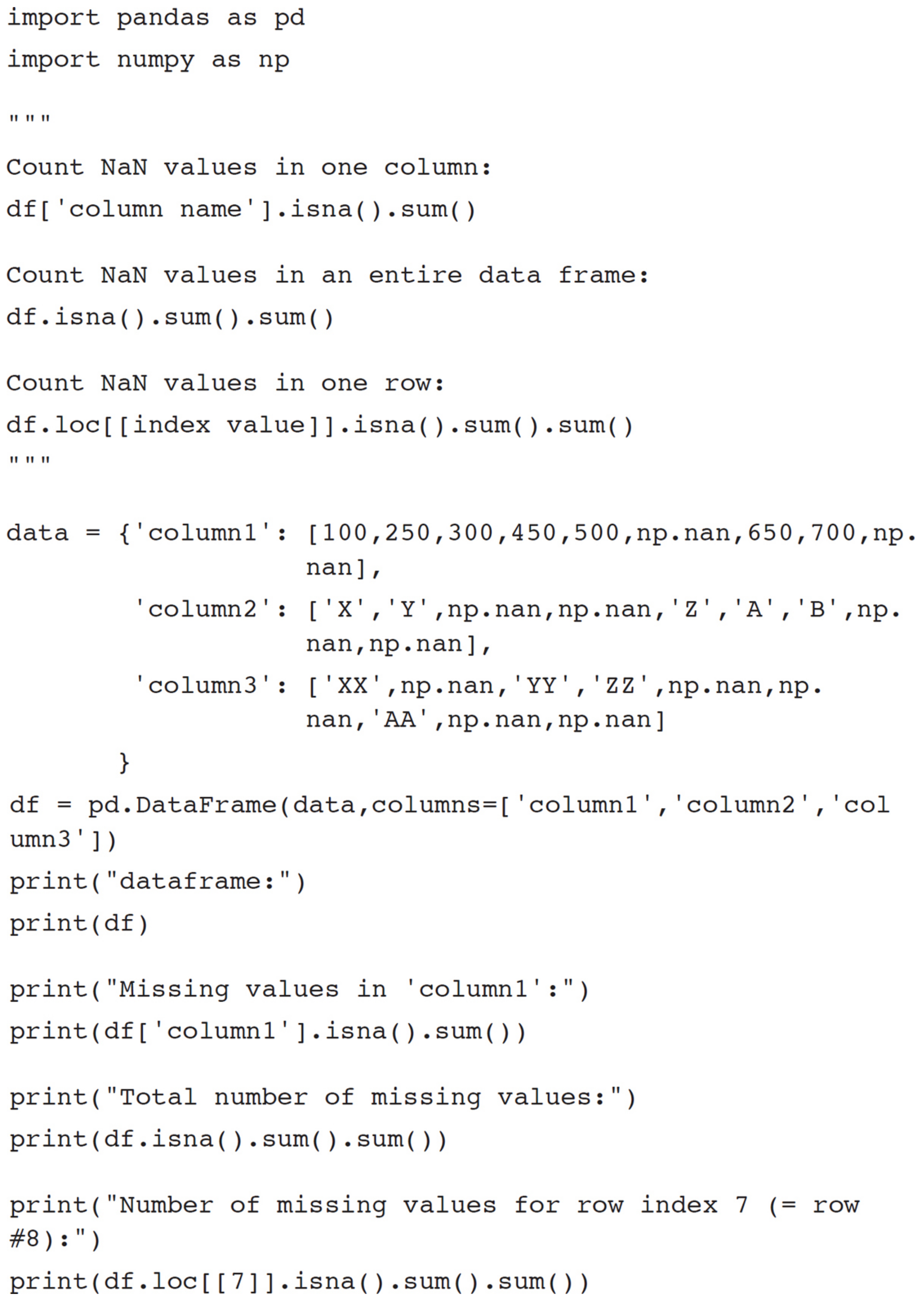
- Count missing data values.
- Remove/drop redundant columns.
- Assign values to missing data.
- Remove duplicate values.
- Check for incorrect values.
- Ensure uniformity of data.
- Use the `Imputer` class to fill with mean, median, and `most_frequent`.
- Assign previous/next value to missing values.
  - random value imputation
  - multiple imputation
  - matching and hot-deck imputation

The following subsections briefly discuss some of the topics in the preceding list, along with some Python-based code samples.

Counting Missing Data Values

Listing 3.11 displays the content of `missing_values2.py` that illustrates how to find the number of missing data values in a `Pandas` data frame.

Listing 3.11: missing_values2.py



```
import pandas as pd
import numpy as np

"""
Count NaN values in one column:
df['column name'].isna().sum()

Count NaN values in an entire data frame:
df.isna().sum().sum()

Count NaN values in one row:
df.loc[[index value]].isna().sum().sum()
"""

data = {'column1': [100,250,300,450,500,np.nan,650,700,np.
                   nan],
        'column2': ['X','Y',np.nan,np.nan,'Z','A','B',np.
                   nan,np.nan],
        'column3': ['XX',np.nan,'YY','ZZ',np.nan,np.
                   nan,'AA',np.nan,np.nan]
       }
df = pd.DataFrame(data,columns=['column1','column2','col
umn3'])
print("dataframe:")
print(df)

print("Missing values in 'column1':")
print(df['column1'].isna().sum())

print("Total number of missing values:")
print(df.isna().sum().sum())

print("Number of missing values for row index 7 (= row
#8):")
print(df.loc[[7]].isna().sum().sum())
```

Listing 3.11 starts with two `import` statements and a comment block that explains the purpose of several `Pandas` methods pertaining sums of values and the `isna()` method for finding `NaN` values in a dataset.

The next portion of Listing 3.11 initializes a dictionary with three arrays of values that are used to initialize the `Pandas` data frame `df`. Next, the missing values in `column1` are displayed, followed by the number of missing values in every column of `df`. The final code block displays the number of missing values for the row whose index is 7. Launch the code in Listing 3.11, and you will see the following output:

```
dataframe:
      column1  column2  column3
0       100.0        X       XX
1       250.0        Y      NaN
2       300.0      NaN       YY
3       450.0      NaN       ZZ
4       500.0        Z      NaN
5         NaN        A      NaN
6       650.0        B       AA
7       700.0      NaN      NaN
8         NaN      NaN      NaN
Missing values in 'column1':
2
Total number of missing values:
11
Number of missing values for row index 7 (= row
#8):
2
```

Navigate to the following webpage, where you will find additional `Python` code samples for data cleaning:

https://lvngd.com/blog/data-cleaning-with-python-pandas/

Drop Redundant Columns

Listing 3.12 displays the content of `drop_columns.py` that illustrates how to remove redundant columns from a Pandas data frame.

Listing 3.12: drop_columns.py

```
import pandas as pd

# specify a valid CSV file here:
df1 = pd.read_csv("my_csv_file.csv") # <= specify
your own CSV file

# remove redundant columns:
df2 = df1.drop(['url'],axis=1)

# remove columns with over 50% missing values
df3 = df2.dropna(thresh=half_count,axis=1)
```

Listing 3.12 initializes the `Pandas` data frame `df1` with the contents of the `CSV` file `my_csv_file.csv` and then initializes the `Pandas` data frame `df2` with the contents of `df1`. It then drops the column `url`, or some other column that exists in your `CSV` file. Finally, the `Pandas` data frame `df3` is initialized with the contents of `Pandas` data frame `df2`, after which columns are dropped if they have more than 50% missing values.

Remove Duplicate Rows

Data deduplication refers to the task of removing row-level duplicate data values. Listing 3.13 displays the content of `duplicates.csv`, and Listing 3.14 displays the content of `duplicates.sh` that removes the duplicate rows and creates the CSV file `no_duplicates.csv`, which contains unique rows.

Listing 3.13: duplicates.csv

```
Male,19,190,0
Male,19,190,0
Male,15,180,0
Male,15,180,0
Female,16,150,0
Female,16,150,0
Female,17,170,0
Female,17,170,0
Male,19,160,0
Male,19,160,0
```

Listing 3.14: remove-duplicates.sh

```
filename1="duplicates.csv"
filename2="no_duplicates.csv"

cat $filename1 | sort |uniq > $filename2
```

Listing 3.14 is straightforward: after initializing the variables `filename1` and `filename2` with the names of the input and output files, respectively, the only remaining code snippet contains Unix pipe ("|") with a sequence of commands. The left-most command displays the contents of the input file, which is redirected to the `sort` command that sorts the input rows. The result of the sort command is redirected to the `uniq` command, which removes duplicates rows, and this result set is redirected to the file specified in the variable `filename2`.

The `sort` command `renders` adjacent duplicate rows, and then the `uniq` command removes adjacent duplicate rows. Launch the code in Listing 3.14, and you will see the output that is displayed in Listing 3.15.

Listing 3.15: no_duplicates.csv

```
Male,19,190,0
Female,16,150,0
Female,17,170,0
Male,15,180,0
Male,19,160,0
Male,19,190,0
Male,19,190,0
```

Display Duplicate Rows

The preceding example shows you how to find the unique rows, and the code sample in Listing 3.16 in this section shows you how to find the duplicate rows.

Listing 3.16: find-duplicates.sh

```
filename1="duplicates.csv"
sorted="sorted.csv"
unique="unique.csv"
multiple="multiple.csv"

# sorted rows:
cat $filename1 | sort > $sorted
# unique rows:
cat $sorted | uniq > $unique
# duplicates rows:
```

```
diff -u $sorted $unique |sed -e '1,3d' -e 's/^ //'
-e 's/-//' > $multiple
```

Listing 3.16 starts by initializing the variables `filename1`, `sorted`, `unique`, and `multiple` to names of CSV files, where only `filename1` is a non-empty file.

The next portion of Listing 3.16 consists of three lines of code that create three text files:

```
sorted.csv
unique.csv
multiple.csv
```

The file `sorted.csv` contains the sorted set of rows from `duplicates.csv`, and the file `unique.csv` contains the unique rows in `sorted.csv`. Therefore, the duplicate rows are the rows that appear in `sorted.csv` that do not appear in `unique.csv`. Launch the code in Listing 3.16, and then inspect the contents of `multiple.csv`.

The third line with the `diff` command generates the list of lines in `$sorted` that are not in `$uniq`, which are of course the duplicate lines. In addition, the output of the `diff` command is redirected to the `sed` command, which does three things:

- Removes the first three text lines.
- Removes an initial space character.
- Removes an initial "-" character.

After the `sed` command has completed, the output is redirected to the file `$multiple` that contains the duplicate rows.

Almost Duplicate Rows

Listing 3.17 displays the contents of the CSV file `people.csv`, and Listing 3.18 shows you how to find a pair of rows (if they exist) in which the same person has different values for height or age (or both).

Listing 3.17: people.csv

```
dawn,slade,42,158
sara,smith,30,160
john,smith,30,170
dave,jones,35,180
john,jones,45,190
sara,smith,30,170
john,smith,32,175
dawn,slade,42,155
```

Listing 3.18: almost_equal_rows.sh

```
filename1="duplicates.csv"

# header row: fname, lname, age, height (cm)
filename="people.csv"

cat $filename | sort | awk -F"," '
BEGIN { arr1[0] = ""; rows = 0; }
{
  arr1[rows] = $0
  rows += 1
}
END {
  for(i=0; i<rows-1; i++) {
    split(arr1[i],row1,",")
    split(arr1[i+1],row2,",")

    if( (row1[1] == row2[1]) && (row1[2] == row2[2]) ) {
      if(row1[3] != row2[3]) {
        print "=> Age mismatch in rows",i,"and",(i+1)
        print row1[1],row1[2],":",row1[3],"and",row2[3]
      }

      if(row1[4] != row2[4]) {
        print "=> Height mismatch in rows",i,"and",(i+1)
        print row1[1],row1[2],":",row1[4],"and",row2[4],"
        centimeters"
      }
    }
  }
}
'
```

Listing 3.18 starts by initializing the variable `filename` with the name of the `CSV` files in Listing 3.17, followed by the `cat` command that pipes the contents of the `CSV` file to the `awk` command that consists of a `BEGIN` block, an execution block, and an `END` block. The `BEGIN` block initializes the first entry in the array `arr1` to an empty string and also initializes the variable `rows` to 0.

The main execution block initializes each row of arr1 to each line in the CSV file, and also increments the variable rows. The `END` block contains a loop that iterates through the rows of the array `arr1` During each iteration, the `split()` function splits the current row using a comma (“,”) as a delimiter. The second invocation of the `split(()` function performs the same operation on the next entry in the array arr1 (which is why the loop has an upper limit of `rows-1` instead of `rows`).

The next portion of Listing 3.18 (still inside the loop) contains conditional logic that checks for duplicate values in the `ith` row and the `(i+1)st` row. Two more conditional statements determine whether or not a duplicate value appears in the `age` attribute or the `height` feature, and an appropriate message is displayed. Launch the code in Listing 3.18, and you will see the following output:

```
=> Height mismatch in rows 1 and 2
dawn slade : 155 and 158 centimeters
=> Age mismatch in rows 4 and 5
john smith : 30 and 32
=> Height mismatch in rows 4 and 5
john smith : 170 and 175 centimeters
=> Height mismatch in rows 6 and 7
sara smith : 160 and 170 centimeters
```

Uniformity of Data Values

An example of the uniformity of data involves verifying that the data in a given feature contains the same units measure. For example, the following set of values have numeric values that are in a narrow range but the units of measure are incorrect:

```
50mph
```

```
50kph
100mph
20kph
```

Listing 3.19 displays the content of `same_units.sh` that illustrates how to ensure that a set of strings have the same unit of measure.

Listing 3.19: same_units.sh

```
strings="120kph 100mph 50kph"
new_unit="fps"

for x in `echo $strings`
do
  number=`echo $x | tr -d [a-z][A-Z]`
  unit=`echo $x | tr -d [0-9]`
  echo "initial: $x"
  new_num="${number}${new_unit}"
  echo "new_num: $new_num"
  echo
done
```

Listing 3.19 starts by initializing the variables `strings` and `new_unit`, followed by a `for` loop that iterates through each string in the `strings` variable. During each iteration, the variables `number` and `unit` are initialized with the characters and digits, respectively, in the current string represented by the loop variable `x`.

Next, the variable `new_num` is initialized as the concatenation of the contents of `number` and `new_unit`. Launch the code in Listing 3.19, and you will see the following output:

```
initial: 120kph
```

```
new_num: 120fps

initial: 100mph
new_num: 100fps

initial: 50kph
new_num: 50fps
```

Too Many Missing Data Values

Datasets with mostly `N/A` values, which is to say, 80% or more are `N/A` or `NaN` values, are always daunting, but not necessarily hopeless. As a simple first step, you can drop rows that contain `N/A` values, which might result in a loss of 99% of the data. A variation of the preceding all-or-nothing step for handling datasets with a majority of `N/A` values is as follows:

- Use a kNN imputer to fill missing values in high value columns.
- Drop low priority columns that have > 50% missing values.
- Use a kNN imputer (again) to fill the remaining missing values.
- Try using 3 or 5 as the # of nearest neighbors.

The preceding sequence attempts to prune insignificant data to concentrate on reconstructing the higher priority columns through data imputation. Of course, there is no guaranteed methodology for salvaging such a dataset, so you need some ingenuity as you experiment with datasets containing highly limited data values. If the dataset is highly imbalanced, consider *oversampling* before you drop columns and/or rows.

Categorical Data

Categorical values are discrete and can easily be encoded by specifying a number for each category. If a category has `n` distinct values, then visualize the `nxn` identity matrix: each row represents one of the distinct values.

This technique is called *one-hot encoding*, and you can use the `OneHotEncoder` class in `scikit-learn` by specifying the dataset `X` and also the column index to perform one-hot encoding:

```
from scikit-learn.preprocessing import
OneHotEncoder
ohc = OneHotEncoder(categorical_features = [0])
X = onehotencoder.fit_transform(X).toarray()
```

Since each one-hot encoded row contains one 1 and (n-1) zero values, this technique is inefficient when `n` is large. Another technique involves the `Pandas map()` function, which replaces string values with a single column that contains numeric values. For example, the following code block replaces `Male` and `Female` with 0 and 1, respectively:

```
values = {'Male' : 0, 'Female' : 1}
df['gender'] = df['gender'].map(values)
```

A variation of the preceding is the following code block:

```
data['gender'].replace(0, 'Female',inplace=True)
data['gender'].replace(1, 'Male',inplace=True)
```

Another variation of the preceding code is this code block:

```
data['gender'].replace([0,1],
['Male','Female'],inplace=True)
```

Keep in mind that the `Pandas map()` function converts invalid entries to `NaN`.

Data Inconsistency

Data inconsistency occurs when distinct value are supposed to be the same value, such as “smith” and “SMITH” instead of “Smith.” Another example would be “YES,” “Yes,” “YS,” and “ys” instead of “yes.” In all cases except for “ys,” you can convert all the other strings to lower case, which replaces all the strings with “smith” or “yes,” respectively.

Alternatively, the `Python`-based `Fuzzy Wuzzy` library can be helpful if there are too many distinct values to specify manually. This module identifies strings that are likely to be the same by comparing two strings and generating a numeric value, such that values closer to each other are more likely to represent the same string.

Mean Value Imputation

Listing 3.20 displays the content of `mean_imputation.py` that shows you how to replace missing values with the mean value of each feature.

Listing 3.20: mean_imputation.py

```
import numpy as np
import pandas as pd
import random

filename="titanic.csv"
df = pd.read_csv(filename)

# display null values:
print("=> Initial df.isnull().sum():")
print(df.isnull().sum())
print()

# replace missing ages with mean value:
df['age'] = df['age'].fillna(df['age'].mean())
```

```
"""
Or use median(), min(), or max():
df['age'] = df['age'].fillna(df['age'].median())
df['age'] = df['age'].fillna(df['age'].min())
df['age'] = df['age'].fillna(df['age'].max())
"""

# FILL MISSING DECK VALUES WITH THE mode():
mode = df['deck'].mode()[0]
#df['deck'] = df['deck'].fillna(mode)

print("=> new age and deck values:")
print([df[['deck','age']]])
```

Listing 3.20 starts with several `import` statements, followed by initializing the variable `df` with the contents of the specified `CSV` file. The next code snippet calculates the number of rows with missing values on a column-by-column basis. The next code snippet replaces the missing values with the mean value of the available values, also on a column-by-column basis. In addition, a commented code shows you how invoke the `median()`, `min()`, and `max()` values, respectively, that you can specify instead of the `mean()`, if you wish to do so.

The next code snippet initializes the variable mode with the mode value of the deck feature, followed by a `print()` statement that displays the values for the `deck` and `age` features. Launch the code in Listing 3.20, and you will see the following output:

```
=> Initial df.isnull().sum():
survived      0
pclass        0
sex           0
```

```
age           177
sibsp           0
parch           0
fare            0
embarked        2
class           0
who             0
adult_male      0
deck          688
embark_town     2
alive           0
alone           0
dtype: int64

=> new age and deck values:
[    deck        age
0       C  22.000000
1       C  38.000000
2       C  26.000000
3       C  35.000000
4       C  35.000000
..    ...        ...
886     C  27.000000
887     B  19.000000
888     C  29.699118
889     C  26.000000
890     C  32.000000

[891 rows x 2 columns]]
```

Random Value Imputation

Random value imputation involves generating random values, which you can use to fill missing values in a dataset. Listing 3.21 displays the content of `random_imputation.py` that shows you how to replace missing values with random values that are selected from within a given feature.

Listing 3.21: random_imputation.py

```
import numpy as np
import pandas as pd
import random

filename="titanic.csv"
df = pd.read_csv(filename)

# display null values:
print("=> Initial df.isnull().sum():")
print(df.isnull().sum())
print()

# replace missing ages with mean value:
df['age'] = df['age'].fillna(df['age'].mean())

#Randomize missing column data
def randomize_deck(df2):
  df = df2.copy()
  data = df["deck"]
  mask = data.isnull()
  samples = random.choices(data[~mask].values , k =
  mask.sum() )
  data[mask] = samples
  return df
```

```
# FILL MISSING DECK VALUES WITH RANDOM non-null
values:
df = randomize_deck(df)

print("=> new age and deck values:")
print([df[['deck','age']]])
```

Listing 3.21 is similar to Listing 3.18: it starts with several `import` statements, followed by initializing the variable `df` with the contents of the specified `CSV` file. The next code snippet calculates the number of rows with missing values on a column-by-column basis.

The next code snippet replaces the missing values in the `age` feature with the mean value of the available values. The next code block is the `Python` function `randomize_deck()`, which randomizes the values in the `deck` feature and then returns the modified frame `df`.

After the `Python` function is a code snippet that invokes the `randomize_deck()` function, after which a `print()` statement displays the values for the `deck` and `age` features. Launch the code in Listing 3.21, and you will see the following output:

```
=> Initial df.isnull().sum():
survived       0
pclass         0
sex            0
age          177
sibsp          0
parch          0
fare           0
embarked       2
class          0
```

```
who             0
adult_male      0
deck          688
embark_town     2
alive           0
alone           0
dtype: int64

=> new age and deck values:
[     deck        age
0       D  22.000000
1       C  38.000000
2       C  26.000000
3       C  35.000000
4       B  35.000000
..    ...        ...
886     E  27.000000
887     B  19.000000
888     D  29.699118
889     C  26.000000
890     E  32.000000

[891 rows x 2 columns]]
```

Matching and Hot-Deck Imputation

Hot-deck imputation is performed by randomly selecting another row that has similar values on other variables and use its value in the row that contains missing values.

You can also impute values by using an existing value that appears in a similar data point, which is also used recommendation systems that utilizes user-user collaboration to impute ratings for movies.

For example, suppose that data collected for a dataset involves estimating a risk factor for each data point, where the risk is derived information from an associated document. However, the information might be insufficient to calculate an accurate risk value. One potential solution involves finding the nearest neighbors to the new data point and then calculating the average of the risk values in the nearest neighbors.

Is a Zero Value Valid or Invalid?

In general, replace a missing numeric value with zero is a risky choice: this value is obviously incorrect if the values of a feature are positive numbers between 1,000 and 5,000 (or some other range of positive numbers). For a feature that has numeric values, replacing a missing value with the mean of existing values can be better than the value zero (unless the average equals zero); also consider using the median value. For categorical data, consider using the mode to replace a missing value.

There are situations where you can use the mean of existing values to impute missing values, but not the value zero, and vice versa. As a first example, suppose that an attribute contains the height in centimeters of a set of persons. In this case, the mean could be a reasonable imputation, whereas 0 suffers from the following:

1. It is an invalid value (nobody has height 0).
2. It will skew statistical quantities, such as the mean and variance.

You might be tempted to use the mean instead of 0 when the minimum allowable value is a positive number, but use caution when working with highly imbalanced datasets. As a second example, consider a small community of 50 residents with

1. 45 people have an average annual income of USD 50,000

2. 4 other residents have an annual income of 10,000,000
3. 1 resident has an unknown annual income

Although the preceding example might seem contrived, it is likely that the median income is preferable to the mean income, and certainly better than imputing a value of 0.

As a third example, suppose that a company generates weekly sales reports for multiple office branches, and a new office has been opened, but has yet to make any sales. In this case, the use of the mean to impute missing values for this branch would produce fictitious results. Hence, it makes sense to use the value 0 for missing sales-related quantities, which will accurately reflect the sales-related status of the new branch.

SKEWED DATASETS

This section contains a skewed `CSV` file, a shell script to generate a single output line, and another shell script that splits the preceding "one liner" into rows that contains four columns.

Listing 3.22 displays the content of the `CSV` file `skewed_four_columns.csv`, Listing 3.23 displays the content of `gen_one_line.sh`, and Listing 3.21 displays the content of `skewed_four_columns.sh` that generates output consisting of rows with an equal number of columns.

Listing 3.22: skewed_four_columns.csv

```
survived,pclass,sex,age
0,3,male,22.0,
1,1,female,38.0,1,3,
female,26.0,1,1,female,35.0,
0,3,male,35.0,0,3,
male,23.0,0,1,male,54.0,0,3,male,2.0,1,3,female,27
.0
1,2,
female,14.0,
1,3,female,4.0,
1,1,female,
58.0,
0,3,male,20.0,
0,3,
male,39.0,
0,3,female,14.0,
1,2,female,55.0,0,3,male,2.0,1,2,male,23.0,
0,3,female,31.0,
```

Listing 3.23: gen_one_line.sh

```
filename="skewed_four_columns.csv"
cat $filename |sed "1d" | awk -F"," '{
printf("%s",$0) }'
```

Listing 3.23 uses the `sed` command to delete the first line of its input, which is the contents of the `CSV` file. The output of the `sed` command is sent to the `awk` command, which prints each input line without a linefeed, thereby generating a one-line string of the contents of the `CSV` file. Launch the code in Listing 3.23, and you will see the following output:

```
0,3,male,22.0,1,1,female,38.0,1,3,female,26.0,1,1,
female,35.0,0,3,male,35.0,0,3,male,23.0,0,1,male,5
4.0,0,3,male,2.0,1,3,female,27.01,2,female,14.0,1,
3,female,4.0,1,1,female,58.0,0,3,male,20.0,0,3,mal
```

```
e,39.0,0,3,female,14.0,1,2,female,55.0,0,3,male,2.
0,1,2,male,23.0,0,3,female,31.0,
```

Notice that the shell script `gen_one_line.sh` matches the initial portion of Listing 3.21 that is shown in bold, and that this shell script is not actually used in the solution for this task. The purpose of showing you the output from `gen_one_line.sh` is to ensure that you understand its output (which becomes the input for the second `awk` command in Listing 3.24), which contains the code that splits the input text into lines of text that contain four fields.

Listing 3.24: skewed_four_columns2.sh

```
filename="skewed_four_columns.csv"

cat $filename |sed "1d" | awk -F"," '{ printf("%s",$0) }' |
awk -F"," '
BEGIN { colCount = 4 }
{
  for(i=1; i<=NF; i++) {
     printf("%s,", $i)
     #if(i < colCount) { printf(",")}
     if(i % colCount == 0) { printf("\n") }
  }
}
' | sed -e 's/,$//' -e 's/,$//'
```

Listing 3.24 starts by initializing the variable `filename` with the name of a `CSV` file, followed by the `cat` command that pipes the contents of `filename` to the `sed` command that removes the first line from the file. The result is piped to the `awk` command that prints each input line without the linefeed character, which results in an output string of a single line of text.

The first `awk` command pipes its output to the second `awk` command that prints each field from the input string. Whenever four fields are printed, a newline is also printed so that the output will consists of rows that contain four fields. The output from the second `awk` command contains a trailing comma in each line, along with two consecutive commas in the final output. These extra commas are removed with the following code snippet:

```
sed -e 's/,$//' -e 's/,$//'
```

Launch the code in Listing 3.24, and you will see the following output:

```
0,3,male,22.0
1,1,female,38.0
1,3,female,26.0
1,1,female,35.0
0,3,male,35.0
0,3,male,23.0
0,1,male,54.0
0,3,male,2.0
1,3,female,27.01
2,female,14.0,1
3,female,4.0,1
1,female,58.0,0
3,male,20.0,0
3,male,39.0,0
3,female,14.0,1
2,female,55.0,0
3,male,2.0,1
2,male,23.0,0
3,female,31.0
```

CSV FILES WITH MULTI-ROW RECORDS

This section contains a `CSV` file with multi-row records such that each field is on a separate line (e.g., `survived:0`) instead of comma-separated field values for each record.

The solution is surprisingly simple when we use `awk`: set `RS` equal to the string pattern that separates records. In our case, we need to set `RS` equal to `\n\n`, after which `$0` will contain the contents of each multi-line record. In addition, specify `FS='\n'` so that get each line is treated as a field (i.e., `$1`, `$2`, and so forth).

Listing 3.25 displays the contents of the `CSV` file `multi_line_rows.csv`, and Listing 3.26 displays the contents of `multi_line_rows.sh`.

Listing 3.25: multi_line_rows.csv

```
survived:0
pclass:3
sex:male
age:22.0

survived:1
pclass:1
sex:female
age:38.0

survived:0
pclass:3
sex:male
age:35.0
```

```
survived:1
pclass:3
sex:female
age:27.0
```

Listing 3.26: multi_line_rows.sh

```
filename="multi_line_rows.csv"

cat $filename | awk '
BEGIN { RS="\n\n"; FS="\n" }
{
  # name/value pairs have this format:
  # survived:0 pclass:3 sex:male age:22.0
  split($1,arr,":"); printf("%s,",arr[2]);
  split($2,arr,":"); printf("%s,",arr[2]);
  split($3,arr,":"); printf("%s,",arr[2]);
  split($4,arr,":"); printf("%s\n",arr[2]);
}'
```

The key idea in Listing 3.26 is shown in bold, which specifies the value of `RS` (record separator) as two consecutive line feed characters and then specifies `FS` (field separator) as a linefeed character. The main block of code splits the fields `$1`, `$2`, `$3`, and `$4` based on a colon (“:”) separator, and then prints the second field, which is the actual data value.

Note that `arr[1]` contains the *name* of the fields, such as `survived`, `pclass`, `sex`, or `age`, whereas `arr[2]` contains the *value* of the fields. Launch the code in Listing 3.26, and you will see the following output:

```
0,3,male,22.0
1,1,female,38.0
```

```
0,3,male,35.0
1,3,female,27.0
```

There is one more detail: Listing 3.26 does not display the header line with the names of the fields. Listing 3.27 shows you how to modify Listing 3.26 to generate the header line.

Listing 3.27: multi_line_rows2.sh

```
filename="multi_line_rows.csv"

cat $filename | awk '
BEGIN { RS="\n\n"; FS="\n" ; count=0}
{
  # executed once just to display the header line:
  if(count == 0) {
    count += 1
    split($1,arr,":"); header = arr[1]
    split($2,arr,":"); header = header "," arr[1]
    split($3,arr,":"); header = header "," arr[1]
    split($4,arr,":"); header = header "," arr[1]
    print header
  }

  # name/value pairs have this format:
  # survived:0 pclass:3 sex:male age:22.0
  split($1,arr,":"); printf("%s,",arr[2]);
  split($2,arr,":"); printf("%s,",arr[2]);
  split($3,arr,":"); printf("%s,",arr[2]);
  split($4,arr,":"); printf("%s\n",arr[2]);
}'
```

Listing 3.27 initializes the variable `count` with the value 0, follows by a conditional block of code that constructs the contents of the variable `header` (which will contain the names of the fields) by sequentially concatenating the field names.

The contents of `header` are printed, and since the value of count has been incremented, this block of code is executed only once, which prevents the `header` line from being repeatedly displayed. Launch the code in Listing 3.27, and you will see the following output:

```
survived,pclass,sex,age
0,3,male,22.0
1,1,female,38.0
0,3,male,35.0
1,3,female,27.0
```

Other variations of the preceding code are also possible, such as changing the display order of the fields. Listing 3.28 displays the fields in reverse order: `age`, `sex`, `pclass`, and `survived`.

Listing 3.28: multi_line_rows3.sh

```
filename="multi_line_rows.csv"

cat $filename | awk '
BEGIN { RS="\n\n"; FS="\n" ; count=0}
{
  # fields displayed in reverse order:
  if(count == 0) {
    count += 1
    split($4,arr,":"); header = arr[1]
    split($3,arr,":"); header = header "," arr[1]
    split($2,arr,":"); header = header "," arr[1]
    split($1,arr,":"); header = header "," arr[1]
   print header
  }

  # name/value pairs have this format:
  # survived:0 pclass:3 sex:male age:22.0
  split($1,arr,":"); survived = arr[2];
  split($2,arr,":"); pclass = arr[2];
  split($3,arr,":"); sex = arr[2];
  split($4,arr,":"); age = arr[2];

  # fields displayed in reverse order:
  printf("%s,%s,%s,%s\n",age, sex, pclass,
  survived)
}'
```

Listing 3.28 contains a conditional block of code that constructs the contents of the variable `header` by sequentially concatenating the field names *in reverse order*. The contents of `header` are printed, and since the value of count has been incremented, this block of code is executed only once, which prevents the `header` line from being repeatedly displayed.

The second block of code constructs an output string by initializing the variables `survived`, `pclass`, `sex`, and `age` and then printing them *in reverse order*. Launch the code in Listing 3.28, and you will see the following output:

```
age,sex,pclass,survived
22.0,male,3,0
38.0,female,1,1
35.0,male,3,0
27.0,female,3,1
```

COLUMN SUBSET AND ROW SUBRANGE OF TITANIC CSV FILE

At this point in the chapter, you have enough knowledge to create your own variations with respect to the order in which the fields are displayed. Note that the `CSV` file contains only a subset of the fields in the Titanic `CSV` file.

As a final example for this section, Listing 3.29 displays a subset of the columns and a subrange of the rows in the Titanic dataset consisting of the passengers who survived.

Listing 3.29: titanic-subrange.sh

```
filename="titanic.csv"

cat $filename | awk -F"," '
BEGIN { start_row = 10; end_row=25; survived_count = 0
   print "=> The row range is
from",start_row,"to",end_row,"\n"
}
{
   if(count == 0) {
     count += 1
     header = $3 "," $4 "," $7
     print header
   }

   if(count >= start_row && count <= end_row) {
      if($1 ~ /1/) {
         survived_count += 1
         print $3 "," $4 "," $7
      }
   }
   count += 1
}
END { print "\n=> Number of survivors:",survived_count}
'
```

Listing 3.29 starts with the `cat` command that pipes the contents of a `CSV` file to the `awk` command, which consists of three parts. The first part initializes the variables `start_row`, `end_row`, and `survived_count` appropriately.

The second part executes a conditional code block only once, which increments the `count` variable and also sets and prints the contents of the variable `header`. The second conditional code block processes the range of rows between `start_row` and `end_row`, which in this example involves rows 10 through 25, respectively. For each of these rows, the `survived_count` is incremented and then fields 3, 4, and 7 are printed. The final code snippet in this part also increments the variable `count`, which keeps track of the number of rows that are processed.

The third part prints the number of survivors, which is tracked by the variable `survived_count`. Launch the code in Listing 3.29 and you will see the following output:

```
=> The row range is from 10 to 25

sex,age,fare
female,27.0,11.1333
female,14.0,30.0708
female,4.0,16.7
female,58.0,26.55
female,55.0,16.0
male,,13.0
female,,7.225
male,34.0,13.0
female,15.0,8.0292
male,28.0,35.5

=> Number of survivors: 10
```

DATA NORMALIZATION

Normalization is the process of scaling numeric columns in a dataset so that they have a common scale. In addition, the scaling is performed as follows:

1. scaling values to the range [0,1]

2. without losing information
3. without distorting any differences that exist in the ranges of values

You can perform data normalization via the function `MinMaxScaler()` in the `scikit- learn` library.

Assigning Classes to Data

Listing 3.30 displays the contents of `product_prices.csv`, and Listing 3.31 displays the content of `assign_classes.py` that illustrates how to assign a class value to each row in a dataset.

Listing 3.30: product_prices.csv

```
item,price
product1,100
product2,200
product3,250
product4,300
product5,400
```

Listing 3.31: assign_classes.py

```
import pandas as pd

df = pd.read_csv("product_prices.csv")
print("contents of df:")
print(df)
print()

# define class ranges:
def class_value2(y):
  if y<=100:
    return '(1) 0 - 100'
  elif y<=200:
    return '(2) 100 - 200'
  elif y<=250:
    return '(3) 200 - 250'
  else:
    return '(4) 250+'

def class_value(y):
  if y<=100:
    return '1'
  elif y<=200:
    return '2'
  elif y<=250:
    return '3'
  else:
    return '4'

df['class1'] = df['price'].apply(class_value)
df['class2'] = df['price'].apply(class_value2)

print("contents of df:")
print(df)
```

Listing 3.31 initializes the `Pandas` data frame `df` with the contents of the `CSV` file `product_prices.csv` (shown in Listing 3.30) and displays its contents. The next portion of Listing 3.31 is the `Python` function `class_value2`, which returns a string whose contents are a range of values that are based on the parameter `y`. For example, if `y` at most 100, the function returns the string `(1) 0 - 100`, and similar strings for larger values of `y`.

The next portion of Listing 3.31 is the `Python` function `class_value` that returns a string 1, 2, 3, or 4, depending on the parameter `y`. The last portion of Listing 3.31 initializes the column `class1` and `class2` in `df`, respectively, by invoking the `apply()` method with the `Python` functions `class_value` and `class_value2`. Launch the code in Listing 3.31, and you will see the following output:

```
contents of df:
     item         price
0    sentence lists start with cap   100
1    product2     200
2    product3     250
3    product4     300
4    product5     400

        item  price class1        class2
0   product1    100      1     (1) 0 - 100
1   product2    200      2   (2) 100 - 200
2   product3    250      3   (3) 200 - 250
3   product4    300      4        (4) 250+
4   product5    400      4        (4) 250+
```

Other Data Cleaning Tasks

As a quick review, here is a list of additional tasks that belong to data cleaning that might be relevant to a given dataset:

- Detect outliers/anomalies.
- Resolve missing data.
- Resolve incorrect data.
- Resolve duplicate data.

- Remove hidden control characters (ex: \t, ^L, and ^M).
- Remove HTML tags (ex: <div>, <a>, and so forth).
- Handle diacritical marks.
- Check for gaps in sequences of data.
- Check for unusual distributions.
- Examine the actual data instead of relying on documentation.

HANDLING CATEGORICAL DATA

A feature containing categorical data can suffer from various issues, such as missing data, invalid data, or inconsistently formatted data. The following section discusses examples of inconsistent categorical data followed by a section that discusses how to map categorical data to numeric values.

Processing Inconsistent Categorical Data

This section contains examples of processing inconsistent data values. For features that have very low cardinality, consider dropping those features, and similarly for numeric columns, those with zero or very low variance.

Next, check the contents of categorical columns for inconsistent spellings or errors. For example, suppose that a feature contains the values `M` and `F` (for male and female), along with a mixture of gender-related strings, some of which are in the following list:

```
male
Male
female
Female
m
f
M
```

```
F
```

The preceding categorical values for gender can be replaced with two categorical values (unless you have a valid reason to retain some of the other values). Moreover, if you are training a model whose analysis involves a single gender, then you need to determine which rows (if any) of a dataset must be excluded. Also check categorical data columns for redundant or missing whitespaces.

Check for data values that have multiple data types, such as a numerical column with numbers as numerals and some numbers as strings or objects. Ensure there are consistent data formats (numbers as integers or floating numbers) and that dates have the same format (for example, do not mix a `mm/dd/yyyy` date format with another date format, such as `dd/mm/yyyy`).

Mapping Categorical Data to Numeric Values

Character data is often called *categorical data*, examples of which include people’s names, home or work addresses, and email addresses. Many types of categorical data involve short lists of values. For example, the days of the week and the months in a year involve seven and twelve distinct values, respectively. Notice that the days of the week have a relationship: each day has a previous day and a next day, and similarly for the months of a year.

However, the colors of an automobile are independent of each other: the color red is not “better” or “worse” than the color blue. However, cars of a certain color can have a statistically higher number of accidents, which is of interest to insurance companies, but we will not address this case here.

There are several well-known techniques for mapping categorical values to a set of numeric values. A simple example where you need to perform this conversion involves the gender feature in the Titanic dataset.

This feature is one of the relevant features for training a machine learning model. The gender feature has {`M,F`} as its set of values. As you will see later in this chapter, Pandas makes it easy to convert the pair of values {`M,F`} to the pair of values {`0,1`}.

Another mapping technique involves mapping a set of categorical values to a set of consecutive integer values. For example, the set {`Red, Green, Blue`} can be mapped to the set of integers {`0,1,2`}. The set {`Male, Female`} can be mapped to the set of integers {`0,1`}. The days of the week can be mapped to {`0,1,2,3,4,5,6`}. Note that the first day of the week depends on the country: in some cases, it is Sunday and in other cases, it is Monday.

Another technique is called *one-hot encoding*, which converts each value to a *vector* (check Wikipedia if you need a refresher regarding vectors). Thus, {`Male, Female`} can be represented by the vectors `[1,0]` and `[0,1]`, and the colors {`Red, Green, Blue`} can be represented by the vectors `[1,0,0], [0,1,0]`, and `[0,0,1]`.

If you vertically line up the two vectors for gender, they form a 2x2 identity matrix, and doing the same for the colors {`R,G,B`} will form a 3x3 identity matrix, as shown here:

```
[1,0,0]
[0,1,0]
[0,0,1]
```

This technique generalizes in a straightforward manner: if you have `n` distinct categorical values, you can map each of those values to one of the vectors in an `nxn` identity matrix.

As another example, the titles in a set {`"Intern","Junior","Mid-Range","Senior", "Project`

`Leader","Dev Manager"`} have a hierarchical relationship in terms of their salaries (which can also overlap, but we'll gloss over that detail for now).

Another set of categorical data involves the season of the year: {`"Spring", "Summer", "Autumn", "Winter"`}, and while these values are generally independent of each other, there are cases in which the season is significant. For example, the values for the monthly rainfall, average temperature, crime rate, and foreclosure rate can depend on the season, month, week, or even the day of the year.

If a feature has a large number of categorical values, then a one-hot encoding will produce many additional columns for each datapoint. Since the majority of the values in the new columns equal 0, this can increase the sparsity of the dataset, which in turn can result in more overfitting and hence adversely affect the accuracy of machine learning algorithms that you adopt during the training process.

Another solution is to use a sequence-based solution in which N categories are mapped to the integers 1, 2, . . ., N. Another solution involves examining the row frequency of each categorical value. For example, suppose that N equals 20, and there are 3 categorical values for 95% of the values for a given feature. You can try the following:

1. Assign the values 1, 2, and 3 to those three categorical values.
2. Assign numeric values that reflect the relative frequency of those categorical values.
3. Assign the category "OTHER" to the remaining categorical values.
4. Delete the rows whose categorical values belong to the 5%.

Mapping Categorical Data to One-Hot Encoded Values

Listing 3.32 displays the content of `one_hot_encode.py` that illustrates how to determine perform one-hot encoding on a CSV file.

Listing 3.32: one_hot_encode.py

```
import pandas as pd
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

filename="titanic.csv"
df = pd.read_csv(filename)

sns.countplot(x='class',data=df)
#plt.show()

print("=> class values:")
print(df['class'])
print()

# create ohe values:
ohe = pd.get_dummies(df['class'])
print("=> ohe:")
print(ohe)
```

Listing 3.32 starts with several import statements followed by a codes snippet that initializes the Pandas data frame `df` with the contents of the CSV file `titanic.csv`. The next snippet generates a chart that displays the relative frequency of each label (there are three such labels) in the `class` feature.

The next portion of Listing 3.32 displays the set of values in the class feature for the entire CSV file. The last portion of Listing 3.32 invokes the `get_dummies()` method, which generates three new columns that represent a one-hot encoding for the categorical values `First`, `Second`, and `Third`. Launch the code in Listing 3.32, and you will see the following output:

```
=> class values:
0          Third
1          First
2          Third
3          First
4          Third
           ...
886       Second
887        First
888        Third
889        First
890        Third
Name: class, Length: 891, dtype: object

=> ohe:

       First  Second  Third
0          0       0      1
1          1       0      0
2          0       0      1
3          1       0      0
4          0       0      1
..       ...     ...    ...
886        0       1      0
887        1       0      0
```

```
888          0       0      1
889          1       0      0
890          0       0      1

[891 rows x 3 columns]
```

WORKING WITH CURRENCY

As you know, the format for currency depends on the country, which includes different interpretations for a “,” and “.” in currency (and decimal values in general). For example, `1,124.78` equals “one thousand one hundred twenty-four point seven eight” in the US, whereas `1.124,78` has the same meaning in Europe (i.e., the “.” symbol and the “,” symbol are interchanged).

If you need to combine data from datasets that contain different currency formats, then you probably need to convert all the disparate currency formats to a single common currency format. There is another detail to consider: currency exchange rates can fluctuate on a daily basis, which in turn can affect the calculation of taxes, late fees, and so forth. Although you might be fortunate enough where you will not have to deal with these issues, it is still worth being aware of them.

Let’s start with a simple task of removing currency symbols from a set of currency values in a CSV file, which is the topic of the next section.

Detect Currency Symbols

Listing 3.33 displays the contents of `usa_currency.csv`, and Listing 3.34 displays the content of `detect_symbols.sh` that illustrates how to detect currency-specific symbols in currency values.

Listing 3.33: usa_currency.csv

```
1,234
  1,234
$1,234
$ 1,234
USD1,234
USD 1,234
EUR5,678
EUR 5,678
```

Listing 3.34: detect_symbols.sh

```
import
filename="usa_currency.csv"

cat $filename | awk '
BEGIN {valid=0; dollar=0; usd=0; eur=0; conv=0;}
{
  if( $0 ~ /\$/ ) {
    printf("%10s contains a $ symbol\n",$0)
    dollar += 1
  }
  else if( $0 ~ /USD/ ) {
    printf("%10s contains USD\n",$0)
    usd += 1
  }
  else if( $0 ~ /EUR/ ) {
    printf("%10s contains EUR: conversion
    required\n",$0)
    eur += 1
    conv += 1
  }
  else {
    printf("%10s contains no symbols\n", $0)
    valid += 1
  }
}
END {
  print ""
  printf("*** SUMMARY ***\n")
  printf("%4d strings contain a $ symbol\n",dollar)
  printf("%4d strings contain USD symbol\n",usd)
  printf("%4d strings contain EUR symbol
  (conversion required)\n",eur)
  printf("%4d strings without any symbols\n",valid)
}
'
```

Listing 3.34 starts with a `cat` command that redirects the contents of a `CSV` file to an awk command that contains a `BEGIN` block and an `END` block. The `BEGIN` block contains a sequence of if/else statements that check whether or not $0 matches $, `USD`, or `EUR`, respectively, in order to determine the type of currency in the current input line, after which the appropriate scalar variables are incremented.

The `END` block consists of a set of `print` statements that display the number of occurrences of each type of currency that appears in the input `CSV` file. Launch the code in Listing 3.34, and you will see the following output:

```
    1,234 contains no symbols
    1,234 contains no symbols
   $1,234 contains a $ symbol
  $ 1,234 contains a $ symbol
 USD1,234 contains USD
USD 1,234 contains USD
 EUR5,678 contains EUR: conversion required
EUR 5,678 contains EUR: conversion required

*** SUMMARY ***
  2 strings contain a $ symbol
  2 strings contain USD symbol
  2 strings contain EUR symbol (conversion
  required)
  2 strings without any symbols
```

Detect Currency Symbols

Listing 3.35 displays the content of `remove_symbols.sh` that illustrates how to remove currency-specific symbols in currency values.

Listing 3.35: remove_symbols.sh

```
import
filename="usa_currency.csv"

cat $filename | awk '
BEGIN {valid=0; dollar=0; usd=0; eur=0; conv=0;}
{
  if( $0 ~ /\$/ ) {
    printf("%10s contains a $ symbol: ",$0)
    dollar += 1
    gsub(/\$/, "", $0)
    printf("cleaned value = %6s\n", $0)
  }
  else if( $0 ~ /USD/ ) {
    printf("%10s contains USD symbol: ",$0)
    usd += 1
    gsub(/USD/, "", $0)
    printf("cleaned value = %6s\n", $0)
  }
  else if( $0 ~ /EUR/ ) {
    printf("%10s contains EUR symbol: ",$0)
    eur += 1
    conv += 1
    gsub(/EUR/, "", $0)
    printf("cleaned value = %6s\n", $0)
  }
  else {
    printf("%10s contains no symbols\n", $0)
    valid += 1
  }
}
END {
  print ""
  printf("*** SUMMARY ***\n")
  printf("%4d strings contain a $ symbol\n",dollar)
  printf("%4d strings contain USD symbol\n",usd)
  printf("%4d strings contain EUR symbol
  (conversion required)\n",eur)
  printf("%4d strings without any symbols\n",valid)
}
'
```

Listing 3.35 contains all the code in Listing 3.34, along with the invocation of the `gsub()` function in each conditional code block in the `BEGIN` block. The purpose of the `gsub()` command is to remove the currency symbols. Launch the code in Listing 3.35, and you will see the following output:

```
   1,234 contains no symbols
    1,234 contains no symbols
   $1,234 contains a $ symbol: cleaned value =  1,234
  $ 1,234 contains a $ symbol: cleaned value =  1,234
 USD1,234 contains USD symbol: cleaned value =  1,234
USD 1,234 contains USD symbol: cleaned value =  1,234
 EUR5,678 contains EUR symbol: cleaned value =  5,678
EUR 5,678 contains EUR symbol: cleaned value =  5,678
```

```
*** SUMMARY ***
  2 strings contain a $ symbol
  2 strings contain USD symbol
  2 strings contain EUR symbol (conversion
  required)
  2 strings without any symbols
```

Converting Currency Values

The next code sample shows you how to calculate the value between a pair of currencies and the value of Bitcoin in USD. As an initial step, launch the following command from the command line:

```
pip3 install forex-python
```

Listing 3.36 displays the content of `convert_currency.py` that illustrates how to determine some currency values.

Listing 3.36: convert_currency.py

```
import pandas as pd

from forex_python.converter import CurrencyRates

curr = CurrencyRates()
usd = curr.get_rates('USD')
print("USD currency:",usd)
print()

#convert USD to EURO:
usd2eur = curr.get_rate('USD', 'EUR')
print("usd2eur:", usd2eur)

#rounded = print(round(usd2eur, 3))
#print("rounded:", rounded)

from forex_python.bitcoin import BtcConverter
# Bitcoin in USD:
btc = BtcConverter()
bcoin = btc.get_latest_price('USD')
print("bcoin: ", bcoin)
```

Listing 3.36 starts with an `import` statement and then initializes the variable `curr` as an instance of the `CurrencyRates` class, and then initializes the variable `usd` with a set of currency `USD` currency rates.

The next code snippet initializes the currency exchange rate from `USD` to the EUR currency. The final code block initializes the variable `btc` as an instance of the `BtcConverter` class and then displays the latest Bitcoin price in `USD` currency. Launch the code in Listing 3.36, and you will see the following output:

```
USD currency: {'EUR': 0.8880994671403198, 'JPY':
114.6714031971581, 'BGN': 1.7369449378330375,
'CZK': 21.611900532859682, 'DKK':
6.608081705150977, 'GBP': 0.7415452930728242,
'HUF': 316.50088809946715, 'PLN':
4.067850799289521, 'RON': 4.392984014209592,
'SEK': 9.275133214920071, 'CHF':
0.9213143872113678, 'ISK': 127.708703374778,
'NOK': 8.84884547069272, 'HRK': 6.685168738898757,
'RUB': 76.66412078152754, 'TRY':
13.378685612788633, 'AUD': 1.4092362344582594,
'BRL': 5.290586145648313, 'CAD':
1.2698934280639433, 'CNY': 6.36101243339254,
'HKD': 7.795648312611013, 'IDR':
14321.74955595027, 'INR': 74.77531083481351,
'KRW': 1202.1403197158081, 'MXN':
20.541385435168742, 'MYR': 4.185523978685613,
'NZD': 1.5126110124333927, 'PHP':
51.079928952042636, 'SGD': 1.349822380106572,
'THB': 33.195381882770874, 'ZAR':
15.242717584369451}
```

```
usd2eur: 0.8880994671403198
bcoin: 38515.2783
```

Listing 3.37 displays the content of `convert_currency2.py` that illustrates how to display side-by-side countries and their currency conversion rates.

Listing 3.37: convert_currency2.py

```
import pandas as pd
```

```
df = pd.DataFrame(data=
['1000USD','2000EUR','3000EUR'],columns=['price'])
print("Data set:")
print(df)
print()

cc = pd.DataFrame({'from':['EUR','USD'],'to':
['USD','EUR'],'rate':[1.33,0.75]})
print("Conversion Table:")
print(cc)
print()
```

Listing 3.37 starts with an `import` statement to initialize the Pandas data frame `df` with three column titles, followed by the variable `cc`, which contains the currency conversion rates. The next code snippet displays the contents of `cc` in a tabular format. Launch the code in Listing 3.37, and you will see the following output:

```
Data set:
          price
0        1000USD
1        2000EUR
2        3000EUR

Conversion Table:
      from   to  rate
0      EUR  USD  1.33
1      USD  EUR  0.75
```

Listing 3.38 displays the content of `mixed_currency.csv`, and Listing 3.39 displays the content of `convert_currency3.sh` that illustrates how to replace a “,” with a “.” to ensure that strings have a valid `USD` currency format.

Listing 3.38: mixed_currency.csv

```
product1|1129.95
product2|2110,99
product3|2.110,99
product4|2.110.678,99
product5|1,130.95
```

Listing 3.39: convert_currency3.sh

```
filename="mixed_currency.csv"

echo "=> initial contents:"
cat $filename
echo

echo "=> UPDATED CONTENTS OF CSV FILE:"
awk -F"|" '
BEGIN { modified=0 }
{
  comma = index($2,",")
  period = index($2,".")
  OLD2=$2

  if(comma > 0 && period == 0) {
    gsub(/,/,".",$2)
    modified += 1
    #print "comma(s) but no period:", $2
  }
  else if( comma > period ) {
    # replace right-most "," with "Z"
    gsub(/,/,"Z",$2)
    # replace "." with ","
    gsub(/\./,",",$2)
    # replace "Z" with "."
    gsub(/Z/,".",$2)
    modified += 1
    #print "comma(s) before period:", $2
  }
  NEW2=$2

  printf("OLD: %18s NEW: %15s\n",OLD2, NEW2)
}
END { print "=> Modified lines:",modified }
' < mixed_currency.csv
```

Listing 3.39 starts by initializing the variable `filename` as `mixed_currency.csv` and then displays its contents. The next portion of Listing 3.39 is an `awk` script that initializes the variables `comma` and `period` with the index of a comma (",") and period (".") for every input line from the file `mixed_currency.csv`. Unlike other programming languages, there is no explicit loop keyword in the code: instead, it is an implicit aspect of the `awk` programming language.

The next block of conditional code checks for the presence of a comma and the absence of a period: if so, then the `gsub()` function replaces the comma (",") with a period (".") in the second field, which is the numeric-portion of each line in Listing 3.38, and the variable modified is incremented. For example, the input line `product3|2110,99` is processed by the conditional block and replaces the content of `$2`, which is the second field, with the value `2110.99`.

The next portion of code checks for the presence of a comma and a period where the location of the comma is on the *right side* of the period: if so, then three substitutions are performed. First, the right-most comma is replaced with the letter `Z`, after which the period is replaced with a comma, and then the letter `Z` is replaced with a period. Launch the code in Listing 3.39 with the following code snippet:

```
./convert-currency3.sh
```

You will see the following output:

```
=> INITIAL CONTENTS OF CSV FILE:
product1|1129.95
product2|2110,99
product3|2.110,99
product4|2.110.678,99
product5|1,130.95

=> UPDATED CONTENTS OF CSV FILE:
OLD:        1129.95 NEW:          1129.95
OLD:        2110,99 NEW:          2110.99
OLD:       2.110,99 NEW:         2,110.99
OLD:        2.110.678,99     2,110,678.99
                    NEW:
OLD:       1,130.95 NEW:         1,130.95
=> Modified lines: 3
```

WORKING WITH DATES

The format for calendar dates varies among different countries, and this belongs to something called the *localization* of data (not to be confused with `i18n`, which is a short-hand term for internationalization). Some examples of date formats are shown here (and the first four are probably the most common):

```
MM/DD/YY
MM/DD/YYYY
DD/MM/YY
DD/MM/YYYY
YY/MM/DD
M/D/YY
D/M/YY
YY/M/D
MMDDYY
DDMMYY
YYMMDD
```

If you need to combine data from datasets that contain different date formats, then converting the disparate date formats to a single common date format will ensure consistency.

The next section shows you how to check for dates in a DD-MM-YYYY format that are out of range. However, this code sample does not check for leap years.

Find Out of Range Dates

Listing 3.40 displays the contents of `dates.txt`, and Listing 3.41 displays the content of `out_of_range.sh` that checks for dates that are out of range (but not leap years).

Listing 3.40: dates.txt

```
12-28-2022
02-28-2022
```

```
05-13-2021
13-11-2023
13-32-2024
```

Listing 3.41: out_of_range.sh

```
cat dates.txt | awk -F"-" '
BEGIN {
  days["01"] = 31; days["02"] = 28; days["03"] =
  31;
  days["04"] = 30; days["05"] = 31; days["06"] =
  30;
  days["07"] = 31; days["08"] = 31; days["09"] =
  30;
  days["10"] = 31; days["11"] = 30; days["12"] =
  31;
}
{
  month = $1; day = $2; year = $3;
  if( (day <= days[month]) && (month <= 12) ) {
    print $0,"is a valid date"
  } else {
    print $0,"is an INVALID date"
  }
}
'
```

Listing 3.41 starts by invoking the `cat` command with the `CSV` file in order to pipe the output to an `awk` command containing a `BEGIN` block the initializes the array variable `days` with the number of days in each month of the year.

Next, the main execution block initializes the variables month, day, and year with the contents of $1, $2, and $3, respectively. Then a conditional code block checks whether or not the day value is at most the value specified in days[month] and also that the month value is at most 12: if true then the date is valid, otherwise the date is invalid. Launch the code in Listing 3.35, and you will see the following output:

```
12-28-2022 is a valid date
02-28-2022 is a valid date
05-13-2021 is a valid date
13-11-2023 is an INVALID date
13-32-2024 is an INVALID date
```

Listing 3.42 displays the contents of `dates2.txt`, and Listing 3.43 displays the content of `out_of_range2.sh` that checks for dates that are out of range, including leap years.

Listing 3.42: dates2.txt

```
02-28-1900
02-29-1900
02-28-2000
02-29-2000
02-28-2020
02-29-2020
02-28-2022
02-29-2022
05-13-2021
13-11-2023
13-32-2024
```

Listing 3.43: out_of_range2.sh

```
cat dates2.txt | awk -F"-" '
function leap_year(year){
  # centuries that are not multiples of 400 are not
  leap years
  # return 0 for leap years and return 1 for non-
  leap years
  if(year % 4 == 0) {
    if(year % 100 == 0) {
      if(year % 400 == 0) return 0
      else return 1
    } else {
      return 0
    }
  } else {
    return 1
  }
}
BEGIN {
  days["01"] = 31; days["02"] = 28; days["03"] =
  31;
  days["04"] = 30; days["05"] = 31; days["06"] =
  30;
  days["07"] = 31; days["08"] = 31; days["09"] =
  30;
  days["10"] = 31; days["11"] = 30; days["12"] =
  31;
}
{
  month = $1; day = $2; year = $3;

  # check for leap year:
```

```
  leap = leap_year(year)
  if ( leap == 0 ) days["02"] = 29
  if ( leap == 1 ) days["02"] = 28
  if( (month <= 12) && (day <= days[month]) ) {
    print $0,"<= valid date"
  } else {
    print $0,"<= INVALID date"
  }
}
'
```

Listing 3.43 contains the same code as Listing 3.42, along with new code blocks, starting with the function `leap_year()` that checks whether or not a given positive integer is a leap year. The actual code is an implementation of the comments at the beginning of the function definition.

Another new code block is defined in the main execution block, which invokes the `leap_year()` function to determine whether or not the current year is a leap year, after which the value of `days["02"]` is updated appropriately. Launch the code in Listing 3.43, and you will see the following output:

```
12-28-2022 is a valid date
02-28-2022 is a valid date
05-13-2021 is a valid date
13-11-2023 is an INVALID date
13-32-2024 is an INVALID date
```

Find Missing Dates

Listing 3.44 displays the content of `pandas_missing_dates.py` that illustrates how to display missing dates from a range of dates.

Listing 3.44: pandas_missing_dates.py

```
import pandas as pd

# A dataframe from a dictionary of lists
data = {'Date': ['2022-01-18', '2022-01-20','2022-
01-21','2022-01-24'],
      'Name': ['Joe', 'John', 'Jane', 'Jim']}
df = pd.DataFrame(data)

# Setting the Date values as index:
df = df.set_index('Date')

# to_datetime() converts string format to a
DateTime object:
df.index = pd.to_datetime(df.index)

start_d="2022-01-18"
end_d="2022-01-25"

# display dates that are not in the sequence:
print("MISSING DATES
BETWEEN",start_d,"and",end_d,":")
dates = pd.date_range(start=start_d,
end=end_d).difference(df.index)

for date in dates:
  print("date:",date)
print()
```

Listing 3.44 starts with an `import` statement and then initializes the variable `data` as a dictionary of lists, which is used to initialize the Pandas data frame `df`. The next code snippet sets the field `Date` as the index, followed by converting the strings in `df` to `DateTime` objects, as shown here:

```
df.index = pd.to_datetime(df.index)
```

Then the variables `start_d` and `end_d` are initialized as string-based dates that represent a date range, after which the variable `dates` is initialized with a set of missing dates. The final loop displays the missing dates. Launch the code in Listing 3.35, and you will see the following output:

```
MISSING DATES BETWEEN 2022-01-18 and 2022-01-25 :
date: 2022-01-19 00:00:00
date: 2022-01-22 00:00:00
date: 2022-01-23 00:00:00
date: 2022-01-25 00:00:00
```

Find Unique Dates

Listing 3.45 displays the contents of `multiple_dates.csv`, and Listing 3.46 displays the content of `pandas_misc1.py` that determines the unique years in Listing 3.45.

Listing 3.45: multiple_dates.csv

```
"dates","values"
2021-01-31,40
2021-02-28,45
2021-03-31,56
2021-04-30,NaN
2022-05-31,NaN
2022-06-30,140
2022-07-31,95
```

```
2022-08-31,40
2023-09-30,55
2023-10-31,NaN
2023-11-15,65
```

Listing 3.46: pandas_misc1.py

```
import pandas as pd

df = pd.read_csv('multiple_dates.csv',
parse_dates=['dates'])
print("df:")
print(df)
print()

the_years = df['dates']
year_list = set(the_years)
arr1 = np.array([])
for long_year in year_list:
  year = str(long_year)
  short_year = year[0:4]
  arr1 = np.append(arr1,short_year)

unique_years = set(arr1)
print("unique_years:")
print(unique_years)
print()

unique_arr =
np.array(pd.DataFrame.from_dict(unique_years))
print("unique_arr:")
print(unique_arr)
print()
```

Listing 3.46 starts with an `import` statement and then initializes the Pandas data frame `df` with the contents of the CSV file `multiple_dates.csv`. After initializing the variable `the_years` with the `dates` feature in `df`, and also initializing `year_list` with the distinct values in the variable `the_years`, a loop extracts the year value for each date and appends this value to the NumPy array `arr1`.

Next, `unique_years` is populated with the distinct values in the variable `arr1` and then displayed. The final code snippet initializes the variable `unique_arr` with the unique years from the dictionary `unique_years`, and the result is displayed. Launch the code in Listing 3.46, and you will see the following output:

```
MISSING DATES BETWEEN 2022-01-18 and 2022-01-25
df:
      dates   values
0  2021-01-     40.0
         31
1  2021-02-     45.0
         28
2  2021-03-     56.0
         31
3  2021-04-      NaN
         30
4  2022-05-      NaN
         31
5  2022-06-    140.0
         30
6  2022-07-     95.0
         31
```

```
7   2022-08-     40.0
          31
8   2023-09-     55.0
          30
9   2023-10-      NaN
          31
10  2023-11-     65.0
          15

unique_years:
{'2022', '2023', '2021'}

unique_arr:
[['2022']
  ['2023']
  ['2021']]
```

Switch Date Formats

Listing 3.47 displays the contents of `standard_formats.csv`, and Listing 3.48 displays the content of `switching_date_formats.sh` that illustrates how to display missing dates from a range of dates.

Listing 3.47: standard_dates.csv

```
2021-01-31
2021-02-28
2022-04-30
2022-05-31
2023-10-31
2023-11-15
```

Listing 3.48: switching_date_formats.sh

```
file="standard_dates.csv"

echo "first output:"
cat $file | awk -F"-" ' { print $3 $2 $1 }'
echo

echo "second output:"
cat $file | awk -F"-" '{ print $3,$2,$1 }'
echo

echo "third output:"
cat $file | awk -F"-" ' { print $3 "-" $2 "-" $1
}'
```

Listing 3.48 starts by initializing the variable `file` with `standard_dates.csv`, whose contents are shown in Listing 3.47. The remaining portion of Listing 3.48 consists of three short code blocks, each of which contains an `awk` command that specifies a hyphen ("-") as the delimiter for the dates. The values of `$1`, `$2`, and `$3` are the three hyphen-delimited values in each input line. For example, `$1`, `$2`, and `$3` equal 2021, 01, and 31, respectively, for the first input line `2021-01-31`.

The first `awk` command displays the date fields in the order `$3`, `$2`, and `$1`. The second first `awk` command also displays the date fields in the order `$3`, `$2`, and `$1`, along with a comma delimiter. The third `awk` command also displays the date fields in the order `$3`, `$2`, and `$1`, along with a hyphen ("-") delimiter. Note that if you want to display the date fields in the order month, day, and year, use the following statement: `print`

`$2,$3,$1`. Launch the code in Listing 3.48, and you will see the following output:

```
first output:
31012021
28022021
30042022
31052022
31102023
15112023

second output:
31 01 2021
28 02 2021
30 04 2022
31 05 2022
31 10 2023
15 11 2023

third output:
31-01-2021
28-02-2021
30-04-2022
31-05-2022
31-10-2023
15-11-2023
```

WORKING WITH QUOTED FIELDS

The code sample in this section shows you how to parse the fields in a `CSV` file that contains a mixture of unquoted fields and quoted fields, in case you want to perform additional processing on each field.

Listing 3.49 displays the contents `quoted_fields1.csv`, and Listing 3.50 displays the content of `quoted_fields1.sh` that illustrates how to parse a CSV file that contains quoted (and possibly empty) fields.

Listing 3.49: quoted_fields1.csv

```
1 2 "5 6" 7 "8 9" A "B C"
"" xer xyz "a b" c "d e" f "g h"
W Z "" "a b" "" cccc "d e" deff
```

Listing 3.50: quoted_fields1.sh

```
file="standard_dates.csv"
filename="quoted_fields1.csv"

echo "Contents of $filename:"
echo "----------------------"
cat $filename
echo "----------------------"
echo ""
```

```
cat $filename | sed 's/""/NaN/g' | awk '
BEGIN { idx = 1; row = 1 }
{
   print "LINE:",$0
   split($0,arr1,"")
   array_len = length(arr1)

   print "LIST OF FIELDS:"
   while(idx<array_len) {
      if(arr1[idx] != "\"") {
        if(arr1[idx] == " ") idx += 1
        while((idx<array_len) && (arr1[idx] != "\"") &&
        (arr1[idx] != " ")){
          printf("%s",arr1[idx])
          idx += 1
        }
        print ""
        # skip quote and a blank space:
        idx += 1
      }

      if(arr1[idx] == "\"") {
        idx += 1
        while((idx<array_len) && (arr1[idx] != "\"") &&
        (arr1[idx] != " ")){
          printf("%s",arr1[idx])
          idx += 1
        }
        # skip quote and a blank space:
        idx += 1
      }
   }
      print "END OF ROW"
      row += 1; idx = 0
   }
   '
```

Listing 3.50 starts by displaying the contents of the `CSV` file in Listing 3.49, followed by an awk statement that prints the current line and then splits the same line into an array variable `arr1` that contains the characters in the current string.

The next portion contains a `while` loop that has two parts with conditional logic. If the current character is *not* a quotation (“”) mark, then a `while` loop prints the characters in the array `arr1` as long as each character is not a quotation (“”) mark or a space. In addition, this `while` loop will terminate if the end of the current line is reached.

The second part checks if the current character *is* a quotation (“”) mark: if so, then a `while` loop prints the characters in the array `arr1` as long as each character is not a quotation (“”) mark or a space. In addition, this `while` loop will terminate if the end of the current line is reached.

After the outer `while` loop terminates, the number of processed records is incremented and the variable `idx` is reset to 0. Launch the code in Listing 3.50, and you will see the following output:

```
Contents of quoted_fields1.csv:
----------------------
1 2 "5 6" 7 "8 9" A "B C"
"" xer xyz "a b" c "d e" f "g h"
W Z "" "a b" "" cccc "d e" deff
----------------------

LINE: 1 2 "5 6" 7 "8 9" A "B C"
LIST OF FIELDS:
1
2
56
```

```
7
89
A
BC
END OF ROW
LINE: NaN xer xyz "a b" c "d e" f "g h"
LIST OF FIELDS:
NaN
xer
xyz
ab
c
de
f
gh
END OF ROW
LINE: W Z NaN "a b" NaN cccc "d e" deff
LIST OF FIELDS:
W
Z
NaN
ab
NaN
cccc
de
deff
END OF ROW
```

Listing 3.51 displays the content `quoted_fields1.py` that uses Pandas to read the contents of the CSV file `quoted_fields1.csv`, which gives you an idea of the parsing that Pandas performs, which in turn involves much less code than the awk script.

Listing 3.51: quoted_fields1.py

```
import pandas as pd

df = pd.read_csv("quoted_fields1.csv")
print("contents of df:")
print(df)
```

Listing 3.51 contains familiar `Pandas` code, and its output is shown below:

```
contents of df:

          1 2 "5 6" 7 "8 9" A "B C"
0     xer xyz "a b" c "d e" f "g h"
1  W Z "" "a b" "" cccc "d e" deff
```

WHAT IS SMOTE?

SMOTE (Synthetic Minority Oversampling Technique) is a technique for synthesizing new samples for a dataset. This technique is based on linear interpolation:

Step 1: Select samples that are close in the feature space.

Step 2: Draw a line between the samples in the feature space.

Step 3: Draw a new sample at a point along that line.

A more detailed explanation of the SMOTE algorithm is here:

- Select a random sample “a” from the minority class.
- Find k nearest neighbors for that example.
- Select a random neighbor “b” from the nearest neighbors.
- Create a line L that connects “a” and “b.”

- Randomly select one or more points “c” on line L.

If need be, you can repeat this process for the other `(k-1)` nearest neighbors to distribute the synthetic values more evenly among the nearest neighbors.

One disadvantage of SMOTE is that the creation of new data points does not take into account the majority class, which could result in some ambiguity if there is overlap between the minority and majority classes. However, variations of SMOTE are more selective about generating synthetic samples, and you can perform an online search about those variations.

The following article discusses aspects of SMOTE, along with an extensive code sample for fraud detection, as well as the use of a GAN (Generative Adversarial Network):

https://towardsdatascience.com/synthetic-data-to-help-fraud-machine-learning-modelling-c28cdf04e12a

The preceding article uses a type of GAN called a CTGAN, and an independent Python-based code sample with CTGAN is here:

https://github.com/koav/CTGAN

One more thing: you will probably need to modify the contents of the notebook `gas.ipynb` (from the preceding Github repository) to include the following code block at the top of the notebook:

```
!pip install table_evaluator --user
!pip install pandas-profiling[notebook]
!pip install ctgan
!pip install sdv
!pip install https://github.com/pandas-
profiling/pandas-profiling/archive/master.zip
```

DATA WRANGLING

Data wrangling means different things to different people, which might cause some confusion unless people clarify what they mean when they talk about data wrangling. Data wrangling involves multiple steps that can include transforming one or more files. Here are some of the interpretations of data wrangling:

- It is part of a sequence of steps.
- Data wrangling transforms datasets.
- It is essentially the same as data cleaning.

This book adopts the approach of the first and second bullet items, but not the third. Navigate to the following webpage, which lists data wrangling as part of a six-step process:

https://en.wikipedia.org/wiki/Data_wrangling

In addition to the steps outlined in the preceding link, data wrangling can also involve the following tasks:

- transforming datasets from one format into another format (convert)
- creating new datasets from subsets of columns in existing datasets (extract)

As you can see, the preceding steps differentiate between converting data to a different format versus extracting data from multiple datasets to create new datasets. The conversion process can be considered a data cleaning task if only the first step is performed; i.e., there is no extraction step.

One additional comment: the interpretation of data wrangling in this chapter is convenient, but it is not a universally accepted standard definition. Hence, you are free to adopt your own interpretation of data wrangling (versus data cleaning), if you find one that better suits your needs.

Data Transformation

In general, data cleaning involves a single data source (not necessarily in a CSV format), with some type of modification to the content of the data source (e.g., filling missing values and changing date formats), without creating a second data source.

For example, suppose that the data source is a MySQL table called `employees` that contains employee-related information. After data cleaning tasks on the `employees` table are completed, the result will still be named the `employees` table. In database terminology, data cleaning is somewhat analogous to executing a `SQL` statement that involves a `SELECT` on a single table.

However, if two CSV files contain different date formats and you need to create a single CSV file that is based on the date columns, then there will be some type of conversion process that could be one of the following:

- Convert the first date format to the second date format.
- Convert the second date format to the first date format.
- Convert both date formats to a third date format.

In the case of financial data, you are likely to also encounter different currencies, which involves a conversion rate between a pair of currencies. Since currency conversion rates fluctuate, you need to decide the exchange rate to use for the data, which can be as follows:

- the exchange rate during the date that the CSV files were generated
- the current currency exchange rate
- some other mechanism

In addition, you might also need to convert the `CSV` files to `XML` documents, where the latter might be required to conform to an `XML` schema, and perhaps also conform to `XBRL`, which is a requirement for business reporting purposes:

https://en.wikipedia.org/wiki/XBRL

Data transformation can involve two or more data sources to create yet another data source whose attributes are in the required format. Here are four scenarios of data transformation with just two data sources A and B, where data from A and from B are combined to create data source C, where A, B, and C can have different file formats:

- all attributes in A and all attributes in B
- all attributes in A and some attributes in B
- a subset of the attributes in A and all attributes in B
- a subset of the attributes in A and some attributes in B

In database terminology, data transformation is somewhat analogous to executing a `SQL` statement that involves a `SELECT` on two or more database tables with a `JOIN` clause. Such `SQL` statements typically involve a subset of columns from each database table, which would correspond to selecting a subset of the features in the data sources.

There is also the scenario involving the *concatenation* of two or more data sources. If all data sources have the same attributes, then their concatenation is straightforward, but you might also need to check for duplicate values. For example, if you want to load multiple `CSV` files into a database table that does not allow duplicates, then one solution involves concatenating the `CSV` files from the command line and then excluding the duplicate rows.

SUMMARY

This chapter started with several Pandas code samples that use Pandas to read CSV files and then calculate statistical values such as the mean, median, mode, and standard deviation of the data values.

Then you learned how to use Pandas to handle missing values in CSV files, starting with CSV files that contain a single column, followed by two-column CSV files.

OceanofPDF.com

CHAPTER 4

WORKING WITH MODELS

This chapter contains an assortment of topics: data scaling, the confusion matrix, feature engineering, model training, feature importance, feature selection, and feature extraction.

The first part of this chapter briefly discusses techniques for scaling data and the importance of performing this task. You will also learn how to scale numeric data via normalization, standardization, and via units of measure.

The second part of this chapter introduces the confusion matrix as well as metrics such as precision, recall, specificity, accuracy, and F1 score. You will also learn about the ROC curve and AUC curve, which are useful for evaluating a trained model.

The third part of this chapter shows you how to train a model using the kNN algorithm with various datasets, including a wine dataset, a BMI dataset, and a diabetes dataset. This section also introduces the SMOTE algorithm for generating synthetic data.

The fourth section discusses feature engineering, which comprises feature selection and feature extraction. The final section discusses how to perform data cleaning, which can involve working with labeled and unlabeled data.

IMPORT STATEMENTS FOR THIS CHAPTER

The following list contains all the `import` statements that you will encounter in the Python code samples for this chapter:

- `from lazypredict.Supervised import LazyClassifier`
- `from scikit-learn.datasets import make_classification`
- `from scikit-learn.ensemble import RandomForestClassifier`
- `from scikit-learn.feature_selection import RFE`
- `from scikit-learn.model_selection import train_test_split`
- `from scikit-learn.preprocessing import StandardScaler`
- `from scikit-learn.tree import DecisionTreeClassifier`
- `from sklearn.datasets import load_breast_cancer`
- `from sklearn.metrics import classification_report`
- `from sklearn.metrics import confusion_matrix`
- `from sklearn.model_selection import train_test_split`

- `from sklearn.neighbors import KNeighborsClassifier`
- `from sklearn.preprocessing import StandardScaler`
- `import matplotlib.pyplot as plt`
- `import numpy as np`
- `import pandas as pd`
- `import seaborn as sns`

TECHNIQUES FOR SCALING DATA

This section and its subsections contain an overview of various scaling techniques for data in a dataset. In a subsequent section, you will see code samples that scale data values.

The following list contains many of the data scaling techniques that you can use to scale data in a dataset:

- Standard Scaler
- Normalizer
- Max-Abs Scaling
- Min-Max Scaling
- Power Transformer
- Quantile Transformation
- Robust Scaler

The first pair of algorithms (Standard Scaler and Normalization) are frequently used for scaling data, and you will see these techniques in many (most?) online code samples.

Normalization rescales the values in the set `X` so that the scaled values lie in the range of `[0,1]` by subtracting `Xmin` from all the values and then dividing all values by `(Xmax-Xmin)`, which are the maximum and minimum values, respectively, in the set `X`.

However, standardization rescales the values in the set `X` by subtracting the mean from all the values and then dividing all values by `std` (the standard deviation), so that the new values are distributed with mean 0 and variance 1.

Although there are multiple algorithms for scaling the values in a dataset, there is no algorithm that is always the "best" choice for scaling the data in a dataset. Moreover, some scaling techniques can influence machine learning classification algorithms, as described in the next section.

Algorithms Influenced by Scaling Data

Algorithms that perform better with scaled data include the SVM (Support Vector Machines), MLP (Multi-Layer Perceptrons), kNN (k Nearest Neighbor), and NB (Naive Bayes). The underlying reason for this improvement is because these algorithms involve a distance metric. These algorithms also perform better when the data has been normalized.

By contrast, Naive Bayes, CART (Classification and Regression Trees), random forests, and LDA (Latent Dirichlet Allocation) are not affected by scaled data values because the associated algorithms do not involve a distance metric: instead, these algorithms use entropy, the Gini impurity, or variance-based techniques to determine the location of data values in their associated structures (such as trees).

EXAMPLES OF SPLITTING AND SCALING DATA

Tasks for cleaning data include checking for invalid data, duplicate data, and missing data. Additional data cleaning tasks can involve currency values and properly formatted dates.

After completing such tasks, you might also need to resize the range of values for individual features, which is called “squashing” the data when it is processed via an activation function in a neural network. Other tasks to consider are described in the following subsections.

Normalize Versus Standardize

Scaling data involves scaling the values of a feature, which can involve *standardizing data* as well as *data normalization*. There are two popular ways to scale data:

- *normalize:* resize value into the range [0,1] by a linear equation
- *standardize:* resize value into the range [0,1] by a Gaussian distribution

The terminology is opposite to what you might expect: *standardizing* data involves a Gaussian distribution, whereas normalizing involves a linear scaling technique. Hence, standardizing data (not normalizing data) involves a normal distribution.

In fact, the formula for scaling data via normalization involves this formula:

`xi' = (xmax - xi)/[xmax - xmin]`

where `xmax` and `xmin` are the maximum and minimum values, respectively, of the numbers in the set {`x1,...,xn`}.

However, scaling data via standardization involves applying a Gaussian distribution to scale the data values. So, although your intuition might tell you that normalizing data involves a Gaussian distribution, the fact is that normalization involves a linear equation instead of the Gaussian distribution (yes, it is confusing).

The following subsections contain more information about normalization and standardization, as well as the importance of splitting data before performing any type of normalization.

One other detail: data normalization is different from database normalization. The latter pertains to various techniques for structuring database tables so that they conform to “normal forms.” Perform an Internet search for detailed information regarding database normalization.

Why Normalize Data?

The previous section describes how to perform normalization of data values versus standardization of data values. The rationale for doing so is simple: data normalization adjusts values of features in a dataset to establish a “level playing field,” which is to say that the values in all the features are in the same general range. Consequently, machine learning algorithms will not be unduly influenced by features that have a significantly larger range of numeric values.

As an example of applying normalization to a dataset, suppose you to want to classify articles of clothing with these features (as well as others):

- the price is in the range of USD 5 to USD 250
- the number of distinct colors range from 1 to 5

Notice that if you scale the price range by dividing each price by 50, then the resulting set of prices are in the same range as the feature pertaining to the colors. Although the price range is a floating point number and the color can be assigned an integer value, the most important detail is that the values are in the same range.

The preceding example might seem contrived because the scaled values have the same lower and upper bound, neither of which is a requirement. If the price values are in the range between 5 and 10, then the algorithm that you choose will perform better than the range between 5 and 250. It is important to ensure that the range of values in different features differ by less than an order of magnitude (i.e., a multiple of 10).

Split Before Normalizing Data

Normalization adjusts data to handle outliers. One technique is min-max normalization, which first subtracts the mean from every value in a dataset, and then divides those values by the maximum value minus the minimum value. The result is a set of values that are between 0 and 1.

Although you might see code samples that perform normalization after other transformations have been applied to a column in a dataset, it is important to perform normalization as the first step. The rationale for doing so is that the training data and the test data will both be in the range between 0 and 1, which in turn prevents data leakage (discussed later).

In addition, avoid randomly splitting groups and avoid randomly splitting data during the training phase.

Scaling Numeric Data via Normalization

A range of values can vary significantly and it is important to note that they often need to be scaled to a smaller range, such as values in the range [-1,1] or [0,1], which you can do via the `tanh` function or the `sigmoid` function, respectively.

For example, measuring a person’s height in terms of meters involves a range of values between 0.50 meters and 2.5 meters (in the vast majority of cases), whereas measuring height in terms of centimeters ranges between 50 centimeters and 250 centimeters: these two units differ by a factor of 100. A person's weight in kilograms generally varies between 5 kilograms and 200 kilograms, whereas measuring weight in grams differs by a factor of 1,000. Distances between objects can be measured in meters or in kilometers, which also differ by a factor of 1,000.

In general, use units of measure so that the data values in multiple features belong to a similar range of values. In fact, some machine learning algorithms require scaled data, often in the range of [0,1] or [-1,1]. In addition to the `tanh()` and `sigmoid()` functions, there are other techniques for scaling data, such as standardizing data (think Gaussian distribution) and normalizing data (linearly scaled so that the new range of values is in (0,1)).

The following examples involve a floating point variable `X` with different ranges of values that will be scaled so that the scaled values are in the interval [0,1].

Example 1: If `X` is in [0,2], then `X`/2 is in the range [0,1].

Example 2: If `X` is in [3,6], then `X`-3 is in the range [0,3], and (`X`-3)/3 is in [0,1].

Example 3: If is `X` is in [-10,20], then `X` +10 is in [0,30], and (`X` +10)/30 is in [0,1].

In general, suppose that `X` is a random variable whose values are in the range `[a,b]`, where `a < b`. You can scale the data values to the range [0,1] by performing two steps:

Step 1: `X-a is in the range [0,b-a]`

Step 2: `(X-a)/(b-a) is in the range [0,1]`

If `X` is a random variable that has the values {`x1, x2, x3, . . ., xn`}, then the formula for normalization involves mapping each `xi` value to `(xi – min)/(max – min)`, where `min` is the minimum value of `X` and `max` is the maximum value of `X`.

As a simple example, suppose that the random variable `X` has the values {`-1, 0, 1`}. Then `min` and `max` are 1 and -1, respectively, and the normalization of {`-1, 0, 1`} is the set of values {`(-1-(-1))/2, (0-(-1))/2, (1-(-1))/2`}, which equals {`0, 1/2, 1`}.

Scaling Numeric Data with Units of Measure

The previous section showed you how to scale numeric values so that they have the same (or similar) range of values. Another option involves inspecting the units of measure for feature values. One example that you have already seen involves a height feature and weight feature in a dataset: in this case, kilograms and centimeters for their units of measure result in numbers that are in a similar range, whereas kilograms and meters would result in different scales of values.

Scaling feature values programmatically as a prelude to training a machine learning model does not alter the feature values in the dataset. However, you could read the dataset into a Pandas data frame, change the unit of measure via conversion, and save the modified data frame to a CSV file. The benefit of performing the preceding steps is that the updated dataset can be processed as-is in existing reports (or new reports) because the new range of feature values are compatible.

Scaling Numeric Data to the Range [a,b]

Given a set of numeric values `X` = {`x1, x2, x3, . . ., xn`} , with a minimum value of `xmin` and a maximum value of `xmax`, and suppose that we want to scale these values so that they are between a and b (a < b). We can perform this transformation in two steps:

Step 1: `xi => (xmax-xi)/(xmax-xmin) * (b-a) = vi`

Step 2: `vi => vi+a = wi`

The `vi` in step 1 are in the range `[0,b-a]` because each `vi` is the product of a rational number between 0 and 1 and the number `b-a`. Step 2 adds the value a to each `vi`, so the resulting set of numbers {`w1, w2, . . . , wn`} are in the range `[a-b]`. Here is a simple example:

```
X = [2, 4, 6, 10]
xmin = 2, xmax=10, a = 16, b=24
{vi} = {(10-2)/8, (10-4)/8, (10-6)/8, (10-10)/8}} * 8
     = {1, 3/4, 1/2, 0} * 8
     = {8, 6, 4, 0}
{wi} = {24, 22, 20, 16}
```

Scaling Numeric Data via Standardization

The standardization technique involves finding the mean `mu` and the standard deviation `sigma`, and then mapping each `xi` value to `(xi – mu)/sigma`. Recall the following formulas:

```
mu = [SUM (x) ]/n
variance(x) = [SUM (x – xbar)*(x-xbar)]/n
sigma = sqrt(variance)
```

As a simple illustration of standardization, suppose that the random variable `X` has the values {`-1, 0, 1`}. Then `mu` and `sigma` are calculated as follows:

```
mu       = (SUM xi)/n = (-1 + 0 + 1)/3 = 0

variance = [SUM (xi- mu)^2]/n
         = [(-1-0)^2 + (0-0)^2 + (1-0)^2]/3
         = 2/3

sigma    = sqrt(2/3) = 0.816 (approximate value)
```

Hence, the standardization of {`-1, 0, 1`} is {`-1/0.816, 0/0.816, 1/0.816`}, which in turn equals the set of values {`-1.2254, 0, 1.2254`}.

As another example, suppose that the random variable `X` has the values {`-6, 0, 6`}. Then `mu` and `sigma` are calculated as follows:

```
mu       = (SUM xi)/n = (-6 + 0 + 6)/3 = 0

variance = [SUM (xi- mu)^2]/n
         = [(-6-0)^2 + (0-0)^2 + (6-0)^2]/3
         = 72/3
         = 24
sigma    = sqrt(24) = 4.899 (approximate value)
```

Hence, the standardization of {`-6, 0, 6`} is {`-6/4.899, 0/4.899, 6/4.899`}, which in turn equals the set of values {`-1.2247, 0, 1.2247`}.

In the preceding two examples, the mean equals 0 in both cases, but the variance and standard deviation are significantly different. One other point: the *normalization* of a set of values *always* produces a set of numbers between 0 and 1.

However, the *standardization* of a set of values can generate numbers that are less than -1 and greater than 1: this will occur when `sigma` is less than the minimum value of every term |`mu - xi`|, where the latter is the absolute value of the difference between `mu` and each `xi` value. In the preceding example, the minimum difference equals 1, whereas `sigma` is 0.816, and therefore the largest standardized value is greater than 1.

The StandardScaler Class

Listing 4.3 displays the content of `standard_scaler.py` that illustrates how to use the `StandardScaler` class to scale values in a dataset.

Listing 4.3: standard_scaler.py

```
from scikit-learn.preprocessing import
StandardScaler

# initialize X_train, X_test, and y_train here:
# include your code here

ssx = StandardScaler()
X_train = ssx.fit_transform(X_train)
X_test = ssx.transform(X_test)

ssy = StandardScaler()
y_train = ssy.fit_transform(y_train)
```

Listing 4.3 contains an `import` statement and then initializes the variable `ssx` as an instance of the `StandardScaler` class of `scikit-learn`. Next, the values in `X_train` and `X_test` are scaled via the methods `fit_transform()` and `transform()`, respectively, of the variable `ssx`. Similarly, the values in `y_train` are scaled via the method `fit_transform()` of the variable `ssy`. Notice that you need to initialize the variables `X_train`, `X_test`, and `y_train` to execute the code in Listing 4.3.

One detail to keep in mind: some machine learning algorithms do *not* require you to scale data, such as the following:

- AdaBoost
- decision trees
- Naive Bayes
- random forests

However, the following machine learning algorithms *require* you to explicitly scale data, such as the following:

- kNN (k Nearest Neighbors)
- Linear Regression
- Logistic Regression
- Neural Networks

Scaling Numeric Data via Robust Standardization

The *robust standardization* technique is a variant of standardization that computes the mean `mu` and the standard deviation `sigma` based on a subset of values. Specifically, this technique uses only the values that are between the 25th percentile and 75th percentile and ignores the first and fourth quartiles, which is where outliers would be located. Let’s define the following variables:

```
X25 = 25th percentile
X75 = 75th percentile
XM = mean of {Xi} values
```

```
XR = robust standardization
```

Then `XR` is computed according to the following formula:

```
XR = (Xi - XM)/(X75 - X25)
```

The preceding technique is also called IQR, which stands for *interquartile range*, and you can see a sample calculation online:

https://en.wikipedia.org/wiki/Interquartile_range

Selecting the Type of Scaling

Feature scaling is important in data preprocessing for various algorithms, some of which are listed here:

- Lasso and Ridge penalties
- distance-based models
- kNNs
- clustering (kMeans)
- SVMs
- ANNs (artificial neural networks)

The preceding algorithms perform better when the predictors have the same scale or within the same boundaries. As a general rule, select a particular type of scaling as follows:

- standardization or robust scaling if outliers exist
- standardization for Gaussian distributions
- normalization for non-normal distributions

Deciding How to Scale Data

Scaling the values of features can improve the quality and predictive power of a model, which would otherwise be biased toward features with larger values. Feature scaling can be performed via normalization or standardization of features. Although we often assume that data is normally distributed, there are exceptions.

Hence, try to ascertain the distribution of the data in the features of a dataset before deciding whether to use either standardization or normalization. For example, if a given feature appears to be uniformly distributed, use normalization (`MinMaxScaler` in `scikit-learn`).

By contrast, for any feature that is approximately Gaussian, use standardization (`StandardScaler`). Again, note that whether you employ normalization or standardization, these are also approximative methods and are bound to contribute to the overall error of the model.

THE CONFUSION MATRIX

A *confusion matrix* provides information that enables you to evaluate classifiers. The confusion matrix is suited for classification tasks: it shows you how many observations were classified by the classification model. In the case of two classes, there are four possibilities:

- True positive
- False positive
- True negative
- False negative

A confusion matrix is easily generated after training a classification-based model in machine learning, as you can see in the following code block:

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("confusion matrix:")
print(cm)
```

In the preceding code block, the variable `y_test` is a one-column vector of target values (often in a CSV file), and `y_pred` is a one-column vector (with the same number of rows as `y_test`) that is generated from a trained model. As an example, consider the following 2x2 confusion matrix and the subsequent description of its contents:

```
[[64  4]
  [ 3 29]]
```

The four values in the preceding 2x2 matrix represent the following quantities:

```
TP = True positive: 64
FP = False positive: 4
TN = True negative: 29
FN = False negative: 3
```

The preceding four quantities occupy the four cells of the following 2x2 binary confusion matrix, whose contents will be discussed in greater detail in a subsequent section:

```
TP | FP
-------
FN | TN
```

Another example of a confusion matrix involves three outcomes, which means that the confusion matrix is 3x3 instead of 2x2:

```
[[12  0  2]
  [ 0 15  1]
  [ 2  0  4]]
```

In addition to 2x2 and 3x3 confusion matrices, an `nxn` confusion matrix is generated when a feature in a dataset consists of `n` different values.

As a practical example, suppose that a dataset that contains clinical trial data for cancer, which involves two classes (healthy and sick). Once again, there are four possible outcomes: true positive, false positive, true negative, and false negative (discussed later). A confusion matrix contains numeric (integer) values for these four quantities. By contrast, linear regression involves terms such as R and R^2 to help you evaluate the accuracy of a model.

Normalized Confusion Matrix

If `cm` is a confusion matrix, such as the confusion matrix in the previous section, the following code snippet normalizes the values in that matrix:

```
cm = cm.astype('float') / cm.sum(axis=1)[:,
np.newaxis]
```

An even simpler way to normalize the values in a confusion matrix `cm` is shown here, where `y_true` are the actual labels in a dataset and `y_pred` are the predicted values that are compared with the actual labels to generate a confusion matrix:

```
cm(y_true, y_pred, normalize='all' )
```

A third way to normalize a confusion matrix involves `scikitplot`, as shown here:

```
import matplotlib.pyplot as plt # dependency for
scikit-plot
import scikitplot as skplt
skplt.metrics.plot_confusion_matrix(Y_TRUE,Y_PRED,
normalize=True)
```

The value for `normalize` in the preceding code snippet might also depend on the version of Python that you have installed on your machine.

Using the confusion matrix from the previous section, the corresponding normalized confusion matrix is here:

```
|0.65 0.40|
|0.11 0.20|
```

A Python Code Sample of a Confusion Matrix

Listing 4.1 displays the content of the Python file `confusion_matrix.py` that shows you how to generate a confusion matrix from a set of numeric data values.

Listing 4.1: confusion_matrix.py

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix

data = {'y_true': [1, 0, 0, 1, 0, 1, 0, 0, 1, 0,
1, 0],
      'y_pred': [1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0,
      0]}

print("=> Data Values:")
print(data)
print()

df = pd.DataFrame(data, columns=
['y_true','y_pred'])
print("=> DataFrame df:")
print(df)
print()
```

```
cm = pd.crosstab(df['y_true'], df['y_pred'],
rownames=['Actual'], colnames=['Predicted'])
print ("=> Confusion matrix:")
print (cm)
print()

cm2 = confusion_matrix(data['y_true'],
data['y_pred'], normalize='all')
print ("=> Normalized Confusion matrix:")
print (cm2)

sns.heatmap(cm2, annot=True)
plt.show()
```

Listing 4.1 starts with `import` statements and then initializes the variable `data` with a set of 0s and 1s for the `y_true` and the `p_pred` elements. These values were arbitrarily selected, so there is no significance to the chosen values (feel free to specify different values).

The next code block initializes the data frame `df` with the values in the variable `data`, after which the confusion matrix `cm` is generated based the data values in `df`. The confusion matrix is printed, and then a second normalized confusion matrix `cm2` is created and also printed. The last code snippet generates and then displays a Seaborn heat map based on the contents of the confusion matrix `cm2`. Launch the code in Listing 4.1, and you will see the following output:

```
=> Data Values:
{'y_true': [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1],
  'y_pred': [1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0]}

=> DataFrame df:
   y_true y_pred
0       0     1
1       1     1
2       0     0
3       1     1
4       0     0
5       1     1
6       1     1
7       1     0
8       1     1
9       1     0
10      1     0
11      1     0

=> Confusion matrix:
Predicted  0  1
Actual
0          2  1
1          4  5

=> Normalized Confusion matrix:
[[0.16666667 0.08333333]
 [0.33333333 0.41666667]]
```

Figure 4.1 displays the heat map generated via the `Seaborn` package, using the data from the confusion matrix `cm2`.

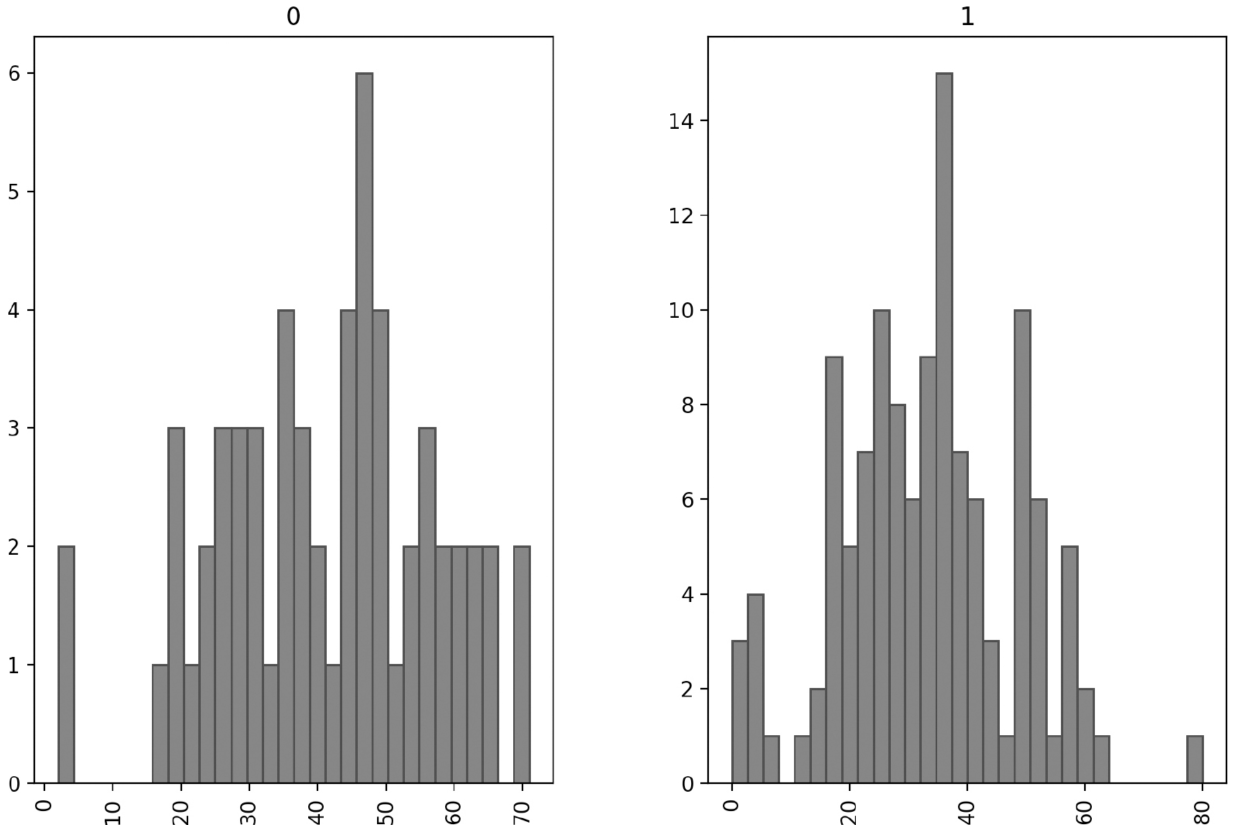


FIGURE 4.1 A best-fitting distribution for a set of random values

What are TP, FP, FN, and TN?

A *binary confusion matrix* (also called an error matrix) is a type of contingency table with two rows and two columns that contains the number of false positives, false negatives, true positives, and true negatives. Here is a 2x2 confusion matrix shown again for your convenience:

```
TP | FP
-------
FN | TN
```

The four entries in the preceding 2x2 confusion matrix have labels with the following interpretation:

- `TP: True Positive`

- `FP: False Positive`
- `TN: True Negative`
- `FN: False Negative`

Just to be sure it is clear, the four entries in the confusion matrix can be described as follows:

- `True Positive (TP): Predicted True and actually True`
- `True Negative (TN): Predicted False and actually False`
- `False Positive (FP): Predicted True and actually False`
- `False Negative (FN): Predicted False and actually True`

Hence, the values on the main diagonal of the confusion matrix are *correct* predictions, whereas the off-diagonal values are *incorrect* predictions. In general, a lower `FP` (false positive) value is better than a `FN` (false negative) value. For example, an `FP` indicates that a healthy person was incorrectly diagnosed with a disease, whereas an `FN` indicates that an unhealthy person was incorrectly diagnosed as healthy.

The confusion matrix can be an `nxn` matrix and not just a 2x2 matrix. For example, if a class has five possible values, then the confusion matrix is a 5x5 matrix, and the numbers on the main diagonal are the true positive results.

Type I and Type II Errors

A Type I error is a false positive, which means that something is erroneously classified as positive when it is negative. However, a Type II error is a false negative, which means that something is erroneously classified as negative when it is positive.

For example, a woman who is classified as pregnant even though she is not pregnant is a Type I error. By contrast, a woman who is classified as *not* pregnant even though she *is* pregnant is a Type II error.

As another example, a person who is classified as having cancer even though that person is healthy is a Type I error. By contrast, a person who is classified as healthy even though that person has cancer is a Type II error.

Based on the preceding examples it is clear that Type I and Type II are not symmetric in terms of the consequences of their misclassification: sometimes it is a case of life-and-death classification. Among the four possible outcomes, the sequence of outcomes, from best to worst, would be the following:

1. True Negative
2. False Positive
3. True Positive
4. False Negative

Although #3 and #4 are both highly undesirable, the third option provides accurate information that people can take appropriate action, whereas the fourth option delays the time at which people can take the necessary precautions. Keep in mind another point regarding a false positive diagnosis: people who are erroneously diagnosed with leukemia or cancer (or some other life threatening disease) might be needlessly subjected to chemotherapy, which has an unpleasant set of consequences.

Accuracy and Balanced Accuracy

You will often see models evaluated via their accuracy, which is defined by the following formula:

```
accuracy    = % of correct predictions
            = (TP + TN) / total cases

balanced accuracy = (recall+specificity)/2
(intermediate)
```

The formula for balanced accuracy involves `recall` and `specificity`, both of which are discussed later. Although accuracy can be a useful indicator, `accuracy` has limited (and perhaps misleading) value for imbalanced datasets. Accuracy can be an unreliable metric because it yields misleading results in unbalanced datasets. Classes with substantially different sizes are assigned equal importance to both false positive and false negative classifications. For example, declaring cancer as benign is worse than incorrectly informing patients that they are suffering from cancer. Unfortunately, accuracy will not differentiate between these two cases.

A Caveat Regarding Accuracy

Accuracy treats Type I and Type II as though they are equally poor; moreover, data belonging to the majority class tends to be given a true classification, and significantly imbalanced datasets tend to skew results toward the majority class.

As a concrete example, consider a dataset with 1,000 rows in which 1% of the people is sick: hence, 990 people are healthy and 10 people are sick. Now train a model to make predictions on this dataset. The no-code solution is to predict that everyone is healthy, which achieves an accuracy of 99%.

The preceding no-code “solution” is obviously unacceptable because it cannot predict *which* people are sick. Instead of accuracy, consider using one or more of the following:

- Matthews Correlation Coefficient (CCM)
- Cohen’s kappa coefficient
- Student’s t-test (for normally distributed data)
- Mann-Whitney U test (for non-normally distributed data)

In general, it's a good idea to calculate the values for precision, recall, and F1 scores and compare them with the value of the accuracy, and see how the models react to imbalanced data.

As a rule of thumb, use the accuracy metric when both classes are equally important and 80% are in the majority class.

Recall, Precision, Specificity, NPV, and Prevalence

The descriptions for recall, precision, NPV, and specificity are given here:

```
• precision:   TP divided by sum of row 1
• NPV:         TN divided by sum of row 2 (negative predic-
               tive value)
• recall:      TP divided by sum of column 1
• specificity: TN divided by sum of column 2
• accuracy:    main diagonal/[sum of all terms]
```

Another term for recall is *sensitivity*. The definition of sensitivity and another formulation for specificity are shown here:

- Sensitivity (TPR) = probability of a positive test, conditioned on truly being positive.
- Specificity (TNR) = probability of a negative test, conditioned on truly being negative.

The formulas for recall, precision, specificity, NPV, and specificity are given here:

```
precision     = TP / (TP + FP)
recall        = TP / (TP + TN)
specificity   = TN / (TN + FP)
Accuracy      = (TP+TN) / [TP+FP+FN+TN]
NPV           = TN / (TN + FN)
prevalence    = (TP+FN) / [TP+TN+FP+FN]
```

One way that might help you remember these formulas is to think of their denominators as the sum of the values in columns or rows, as shown here:

- `Accuracy = (sum of main diagonal)/(sum-of-four-terms)`
- `Precision = TP/(sum-of-row-one)`
- `Recall = TP/(sum-of-column-one)`
- `Specificity = TN/(sum-of-column-two)`
- `False positive rate = FP/(sum-of-column-two)`

Recall (also called *sensitivity*) is the proportion of the correctly predicted positive values in the set of actually positively labeled samples: this equals the fraction of the positively labeled samples that were correctly predicted by the model.

The following code snippet shows you how to invoke the `recall_score()` method, which provides a labels parameter for multi-class classification:

```
from sklearn.metrics import recall_score
recall_score(y_true, y_pred, labels=
[1,2],average='micro')
```

The following code snippet shows you how to invoke the `precision_score()` method, which provides a labels parameter for multi-class classification:

```
from sklearn.metrics import precision_score
precision_score(y_true, y_pred, labels=
[1,2],average='micro')
```

Another technique that might help you remember how to calculate precision and recall is to notice that

1. both have the same numerator (=TP)
2. the precision denominator is the sum of the first row
3. the recall denominator is the sum of the first column

Thus, we can describe accuracy, recall, precision, and specificity as follows:

- Accuracy is the percentage of correctly classified samples of all the samples.
- Recall is the percentages of correctly classified positives of all actual positives.
- Precision is the percentage of correctly classified positives from all predicted positives.
- Specificity is the proportion of negatively labeled samples that were predicted as negative.
- Prevalence is a fraction of total population that is labeled positive.

In essence, sensitivity reflects the extent to which a given test identifies true positive values, whereas specificity reflects the extent to which a given test identifies true negative values. There is often a trade-off between sensitivity and specificity: a higher value for sensitivity involves a lower value for specificity (and vice versa). An extensive list of formulas for metrics is here:

https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Precision Versus Recall: How to Decide?

Sometimes precision is more important than recall: of the set of cases that were predicted as valid, how many times were they true? If you are predicting books that are suitable for under 18 people, you can afford to reject a few books, but cannot afford to accept bad books. If you are predicting thieves in a supermarket, we need more precision. As you can probably surmise, customer trust will decrease due to false positives.

Precision is the proportion of the samples that are actually positive in the set of positively predicted samples, which is expressed informally as

precision = (# of *correct* positive) / (# of *predicted* positive)

Note: Precision is important when false positives are more important than false negatives, such as spam detection, and you want to minimize FP.

Recall is the proportion of the samples that are actually positive in the set of actual positive samples, which is expressed informally as

recall = (# of *predicted* positive) / (# of *actual* positive)

Note: Recall (a.k.a. sensitivity) is important when false negatives are more important than false positives, such as cancer detection, and you want to minimize FN.

[2] TPR, FPR, PV, FDR, and FOR

The quantities `TPR`, `FPR`, `NPV`, `FDR`, and `FOR` are additional terms that you might encounter, and they are defined in this section.

```
TPR = true positive rate
TPR = proportion of positively labeled samples
that are correctly predicted positive
TPR = TP/[TP+FN] = TP/(sum-of-column-one)

FPR = false positive rate
```

FPR = proportion of negatively labeled samples
that are *incorrectly* predicted positive
FPR = FP/[TN+FP] = FP/(sum-of-column-two)

NPV = Negative Predictive Value or NPV
NPV = proportion of negatively labeled samples
that are *correctly* predicted negative
NPV = TN/[TN+FN] = TN/(sum-of-row-two)

FDR = false discovery rate = 1 - PPV = FP/[TP+FP]
= FP/(sum-of-row-one)

FOR = false omission rate
FOR = 1 - NPV = FN/[TN+FN] = FN/(sum-of-row-two)

The following list contains the values of the quantities `TPR`, `FPR`, `NPV`, `FDR`, and `FOR`:

- `TPR = TP/(sum-of-column-one)`
- `FPR = FP/(sum-of-column-two)`
- `NPV = TN/(sum-of-row-two)`
- `FDR = FP/(sum-of-row-one)`
- `FOR = FN/(sum-of-row-two)`

Earlier in this chapter, you learned about a confusion matrix, and the following output shows you the calculated values for precision, recall, F1-score, and accuracy that can be programmatically generated via the `classification_report` class in `sklearn.metrics`, as shown here:

```
              precision    recall  f1-score   support

           0       0.96      0.94      0.95        68
           1       0.88      0.91      0.89        32

    accuracy                           0.93       100
   macro avg       0.92      0.92      0.92       100
weighted avg       0.93      0.93      0.93       100
```

THE ROC CURVE AND AUC CURVE

ROC is an acronym for Receiving Operator Characteristics, and a ROC curve plots the performance of a model by displaying the FP (false positive) rate on the horizontal axis and the TP (true positive) rate on the vertical axis. Note that the `TN` (the true negative rate) is also called the *specificity*.

The area under the ROC curve (abbreviated as ROC AUC) assesses overall classification performance. If the ROC curve is on top of the dashed line, the AUC is 0.5 (half of the square area), and it means the model result is no different from a completely random draw. If the ROC curve is very close to the northwest corner, the AUC will be close to 1.0.

The ROC curve provides a visual comparison of classification models that shows the trade-off between the true positive rate and the false positive rate. Both axes have values between 0 and 1: the vertical axis is the true positive rate (`TPR`) whereas the horizontal axis is the false positive rate (`FPR`). The ROC curve provides a view of model performance at different threshold values.

The goal is to increase `TPR` while simultaneously maintaining a low `FPR`; however, both values increase together, so it is a question of the tolerance level for false positives. After selecting one class as positive and another class as negative in a binary classification task, launch your code and then display a confusion matrix as well as the values for TP, FP, FN, and TN by including the following type of code:

```
# generate the confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("confusion matrix:")
print(cm)
```

```
from sklearn.metrics import confusion_matrix,
classification_report
print(classification_report(y_test, y_pred))
```

The preceding code block generates the following type of output (the numbers depend on the dataset):

```
[[64  4]
 [ 3 29]]
              precision    recall  f1-score   support

           0       0.96      0.94      0.95        68
           1       0.88      0.91      0.89        32

    accuracy                           0.93       100
   macro avg       0.92      0.92      0.92       100
weighted avg       0.93      0.93      0.93       100
```

What is the AUC Curve?

AUC is the area under ROC curve between (0,0) and (1,1), which aggregates the performance of the model at all threshold values. The area under the ROC curve (ROC AUC) is a measure of the accuracy of the model. Models closer to the diagonal are less accurate and the models with perfect accuracy will have an area of 1.0.

The AUC is a value between 1.0 (excellent fit) to 0.5 (random draw). The predictability of a model can be considered “excellent” if the AUC is more than 0.9, and “good” if the AUC is above 0.8.

The best possible value of AUC is 1 (a perfect classifier) and the worst value is 0 (if all the predictions are wrong). The AUC is independent of the classification threshold value.

Calculating ROC AUC Values

The Python-based `lazypredict` module is an open source module that calculates statistics quantities after training a dataset on multiple algorithms, and its home page is here:

https://pypi.org/project/lazypredict/

Listing 4.2 displays the content of the Python file `lazypredict1.py` that shows you how to use the `lazypredict Python` module for calculating values such as ROC AUC and `F1` scores for a given dataset.

Listing 4.2: distfit1.py

```
# pip3 install lazypredict

import pandas as pd
from lazypredict.Supervised import LazyClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import
train_test_split
```

```
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)

df = pd.read_csv("titanic2.csv")
X = df[["age","class"]]
y = df[["survived"]]

X_train, X_test, y_train, y_test =
train_test_split(X, y,test_size=.5,random_state
=123)

classifier =
LazyClassifier(verbose=0,ignore_warnings=True,
custom_metric=None)
models, predictions = classifier.fit(X_train,
X_test, y_train, y_test)
print(models)
```

Listing 4.2 starts with several `import` statements, followed by setting three display-related properties in Pandas. The next portion of code initializes the variable `df` with the contents of `titanic2.csv`, and then the variable `X` is initialized with the `age` feature, whereas the variable `y` is initialized with the survived feature from the data frame `df`.

The next portion of Listing 4.2 invokes the `train_test_split()` method in scikit-learn to initialize the variables `X_train`, `X_test`, `y_train`, and `y_test`. Next, the variable classifier is initialized as an instance of the `LazyClassifier` class that is in the `lazypredict` library. The final code snippet in Listing 4.2 initializes the variables models and predictions with the result of invoking the `fit()` method of the variable classifier. Launch the code in Listing 4.2, and you will see the following output:

```
                              Accuracy  Balanced  ROC AUC    F1   Time
                                        Accuracy            Score  Taken
Model
NuSVC                             0.70      0.68     0.68  0.71   0.02
NearestCentroid                   0.65      0.66     0.66  0.66   0.02
Perceptron                        0.63      0.65     0.65  0.64   0.02
KNeighborsClassifier              0.74      0.63     0.63  0.71   0.02
RandomForestClassifier            0.65      0.61     0.61  0.65   0.16
AdaBoostClassifier                0.70      0.61     0.61  0.68   0.09
GaussianNB                        0.67      0.59     0.59  0.66   0.02
LinearDiscriminantAnalysis        0.73      0.59     0.59  0.68   0.03
LinearSVC                         0.73      0.59     0.59  0.68   0.02
SGDClassifier                     0.74      0.59     0.59  0.68   0.02
LogisticRegression                0.73      0.58     0.58  0.67   0.02
RidgeClassifier                   0.73      0.58     0.58  0.67   0.03
RidgeClassifierCV                 0.74      0.58     0.58  0.67   0.02
XGBClassifier                     0.65      0.58     0.58  0.64   0.04
LabelSpreading                    0.69      0.57     0.57  0.66   0.02
BaggingClassifier                 0.64      0.57     0.57  0.64   0.04
LabelPropagation                  0.69      0.57     0.57  0.66   0.02
QuadraticDiscriminantAnalysis     0.49      0.57     0.57  0.50   0.02
LGBMClassifier                    0.73      0.56     0.56  0.65   0.03
DummyClassifier                   0.62      0.55     0.55  0.62   0.02
DecisionTreeClassifier            0.59      0.55     0.55  0.60   0.02
PassiveAggressiveClassifier       0.68      0.53     0.53  0.62   0.02
CalibratedClassifierCV            0.70      0.53     0.53  0.61   0.03
ExtraTreesClassifier              0.55      0.53     0.53  0.56   0.12
SVC                               0.69      0.51     0.51  0.59   0.02
BernoulliNB                       0.69      0.50     0.50  0.57   0.02
ExtraTreeClassifier               0.53      0.49     0.49  0.54   0.02
```

What is the TOC Curve?

The following webpage contains a Python code sample using SkLearn and the Iris dataset, and also code for plotting the ROC:

https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

The following webpage contains an assortment of Python code samples for plotting the ROC:

https://stackoverflow.com/questions/25009284/how-to-plot-roc-curve-in-python

By contrast, a TOC graph plots the `(TP+FP)-TP` values on the horizontal axis and the `TP` values for the vertical axis. The interesting fact about a TOC graph is that it enables you to determine the confusion matrix for every point in TOC space.

Scoring Rules

A *scoring rule* is a method for evaluating the accuracy of predicted probabilities, which you can perform via log loss (cross entropy) or Brier score.

A log loss score equal to 0 indicates a model with “perfect skill.” In addition, `log_loss` is available in scikit-learn with the following code snippet:

```
from sklearn.metrics import log_loss
```

The `brier` score loss function is also available in scikit-learn with the following code snippet:

```
from sklearn.metrics import brier_score_loss
```

Perform an online search for more information regarding the log loss and Brier score.

ROC AUC and PR AUC

The choice of the ROC AUC versus PR AUC (Precision/Recall AUC) depends on the characteristics of your dataset. In particular, use ROC AUC when both classes are equally important.

However, a highly skewed dataset, or a dataset with a small number of incorrect predictions, can adversely affect both the ROC curve and the ROC AUC curve. In this case, consider using the PR AUC curve, especially when the positive class is more important.

In addition, the PR AUC is well-suited for binary predictions involving imbalanced datasets. The following code snippet shows you how to import `precision_recall_curve` from scikit-learn:

```
from sklearn.metrics import precision_recall_curve
Precision, recall,_ =
precision_recall_curve_ytest, scores)
```

EXPLORING THE TITANIC DATASET

EDA involves a plethora of tasks, such as detecting patterns in the data, detecting outliers/anomalies (if any), and also testing various hypotheses. As a starting point, Listing 4.4 displays the content of `desc_titanic.py` that shows you how to describe the contents of a dataset.

Listing 4.4: desc_titanic.py

```
import pandas as pd

filename="titanic.csv"
train_df = pd.read_csv(filename)
print("Description of",filename)
print(train_df.describe())
```

Listing 4.4 starts with an `import` statement and then initializes the Pandas data frame `train_df` with the contents of the CSV file `titanic.csv`. The final code snippet invokes the `describe()` method to compute the mean, maximum, minimum, and quartile values for each feature and then displays their values. Launch the code in Listing 4.13, and you will see the following output:

```
Description of titanic.csv
        survived     pclass         age       sibsp       parch        fare
count 891.000000 891.000000 714.000000 891.000000 891.000000 891.000000
mean    0.383838   2.308642  29.699118   0.523008   0.381594  32.204208
std     0.486592   0.836071  14.526497   1.102743   0.806057  49.693429
min     0.000000   1.000000   0.420000   0.000000   0.000000   0.000000
25%     0.000000   2.000000  20.125000   0.000000   0.000000   7.910400
50%     0.000000   3.000000  28.000000   0.000000   0.000000  14.454200
75%     1.000000   3.000000  38.000000   1.000000   0.000000  31.000000
max     1.000000   3.000000  80.000000   8.000000   6.000000 512.329200
```

Listing 4.5 displays the content of `info_titanic.py` that shows you how display information about the `Titanic` dataset.

Listing 4.5: info_titanic.py

```
import pandas as pd
```

```
filename="titanic.csv"
train_df = pd.read_csv(filename)
print("Information about",filename)
print(train_df.info())
```

Listing 4.5 starts with an `import` statement and then initializes the Pandas data frame `train_df` with the contents of the CSV file `titanic.csv`. The final code snippet invokes the `info()` method to display the type of each feature and the number of non-null values for each feature and then displays their values. Launch the code in Listing 4.5, and you will see the following output:

```
Information about titanic.csv
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):

#    Column         Non-Null Count  Dtype
---  ------         --------------  -----
0    survived       891 non-null    int64
1    pclass         891 non-null    int64
2    sex            891 non-null    object
3    age            714 non-null    float64
4    sibsp          891 non-null    int64
5    parch          891 non-null    int64
6    fare           891 non-null    float64
7    embarked       889 non-null    object
8    class          891 non-null    object
9    who            891 non-null    object
10   adult_male     891 non-null    bool
```

```
11    deck              203 non-null    object
12    embark_town       889 non-null    object
13    alive             891 non-null    object
14    alone             891 non-null    bool
dtypes: bool(2), float64(2), int64(4), object(7)
memory usage: 92.4+ KB
```

Listing 4.6 displays the content of `null_titanic.py` that shows you how display information about null values in the `Titanic` dataset.

Listing 4.6: null_titanic.py

```
import pandas as pd

filename="titanic.csv"
df = pd.read_csv(filename)

is_null   = df.isnull()
null_sum  = df.isnull().sum()
null_cnt  = df.isnull().count()
null_cnt2 = df.count()

# uncomment this block for more details:
#print("=> df info:")
#print(df.info())
#print()
#print("=> df describe:")
#print(df.describe())
#print()

print("=> Is Null:")
print(is_null)
print()
print("=> Null sum:")
print(null_sum)
print()
print("=> Null count:")
print(null_cnt)
print()
print("=> Null count2:")
print(null_cnt2)
print()
```

Listing 4.6 starts with an `import` statement and then initializes the Pandas data frame `train_df` with the contents of the CSV file `titanic.csv`. The next portion of code initializes the variable `null_sum` that displays True or False, depending on whether or not each cell value is NaN or has a value. Next, the code initializes the variables `null_sum`, `null_cnt`, and `null_cnt2` with the number of NaN values, the number of values in each column, and the count of the non-null values, respectively, in the Titanic dataset. Launch the code in Listing 4.6, and you will see the following output:

```
=> Is Null:
    survived pclass   sex   age ...  deck embark_town alive alone
0      False  False False False ...  True       False False False
1      False  False False False ... False       False False False
2      False  False False False ...  True       False False False
3      False  False False False ... False       False False False
4      False  False False False ...  True       False False False
..       ...    ...   ...   ... ...   ...         ...   ...   ...
886    False  False False False ...  True       False False False
887    False  False False False ... False       False False False
888    False  False False  True ...  True       False False False
889    False  False False False ... False       False False False
890    False  False False False ...  True       False False False

[891 rows x 15 columns]
```

```
=> Null
sum:
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
```

```
alone          0
dtype:
int64

=> Null
count:
survived    891
pclass      891
sex         891
age         891
sibsp       891
parch       891
fare        891
embarked    891
class       891
who         891
adult_male  891
deck        891
embark_town 891
alive       891
alone       891
dtype:
int64

=> Null
count2:
survived       891
pclass         891
sex            891
age            714
sibsp          891
```

```
parch          891
fare           891
embarked       889
class          891
who            891
adult_male     891
deck           203
embark_town    889
alive          891
alone          891
dtype: int64
```

Listing 4.7 displays the content of `mean_titanic.py` that shows you how to calculate mean values in the `Titanic` dataset.

Listing 4.7: mean_titanic.py

```
import pandas as pd

filename="titanic.csv"
train_df = pd.read_csv(filename)

mean_values = train_df.groupby(['sex']).mean()
print("Mean values by Gender:")
print(mean_values)
print()

mean_values2 =
train_df.groupby(['sex','fare']).mean()
print("=> Mean values by Gender and Fare:")
print(mean_values2)
```

Listing 4.7 starts with an `import` statement and then initializes the Pandas data frame `train_df` with the contents of the CSV file `titanic.csv`. The next portion of code initializes the variable `mean_values` with the mean of the values for each gender (i.e., male and female) and then prints the results.

Similarly, the second portion of code initializes the variable `mean_values2` with the mean of the values for each gender as well as for the `fare` feature and then prints the results. Launch the code in Listing 4.7, and you will see the following output:

```
=> Mean values by Gender:
       survived   pclass        age     sibsp     parch       fare   adult_    alone
                                                                       male
sex
female 0.742038 2.159236 27.915709 0.694268 0.649682 44.479818 0.000000 0.401274
male   0.188908 2.389948 30.726645 0.429809 0.235702 25.523893 0.930676 0.712305


=> Mean values by Gender and Fare:
                   survived  pclass   age  sibsp  parch  adult_male  alone
sex      fare
female   6.7500         0.0     3.0  18.0    0.0    0.0         0.0    1.0
         7.2250         1.0     3.0  15.0    0.0    0.0         0.0    1.0
         7.2292         1.0     3.0  13.0    0.0    0.0         0.0    1.0
         7.2500         1.0     3.0  22.0    0.0    0.0         0.0    1.0
         7.4958         1.0     3.0  18.0    0.0    0.0         0.0    1.0
...      ...            ...     ...   ...    ...    ...         ...
male     221.7792       0.0     1.0   NaN    0.0    0.0         1.0    1.0
         227.5250       0.0     1.0   NaN    0.0    0.0         1.0    1.0
         247.5208       0.0     1.0  24.0    0.0    1.0         1.0    0.0
         263.0000       0.0     1.0  41.5    2.0    3.0         1.0    0.0
         512.3292       1.0     1.0  35.5    0.0    0.5         1.0    0.5

[349 rows x 7 columns]
```

This concludes the portion of the chapter pertaining to data cleaning and data wrangling. The next section contains a sequence of tasks that you need to perform to train a model on a dataset using a classification algorithm. Subsequent sections contain Python code samples that show you how to use the kNN algorithm to train a model on various datasets, such as `wine.csv` and later with `bmi.csv` and also with `titanic2.csv`. Hence, you will gain practice with classification algorithms, and you will see how to modify a few lines of code so that you can use several other classification algorithms in addition to the kNN algorithm.

STEPS FOR TRAINING CLASSIFIERS

This section contains the list of steps that you need to perform whenever you want to train a model on a classification task:

- Step 1: Import required Python libraries.
- Step 2: Populate a Pandas data frame `df` from a CSV file.
- Step 3: Specify columns for `X` and `y` (from `df`).
- Step 4: Populate `X_train`, `X_test`, `y_train`, and `y_test`.
- Step 5: Perform feature scaling (`X_train` and `y_train`).
- Step 6: Create an instance `cls` of a specific classifier class.
- Step 7: Invoke the `fit()` method of `cls`.
- Step 8: Invoke the `predict()` method of `cls`.
- Step 9: Populate a confusion matrix cm.
- Step 10: Display the contents of the confusion matrix `cm`.
- Step 11: Inspect values for the precision, recall, and F1 score.

For your convenience, a subsequent section performs all of the steps in the preceding bullet list, which you can use as a template for your own code involving different classification algorithms.

Before we proceed, there are several points to keep in mind. First, you need to perform data cleaning before any of the steps in the preceding bullet list.

Second, the `y` column is the target column, which consists of a single column of values.

Third, `X` is a subset of the columns in the data frame `df`, which can involve a feature selection algorithm or perhaps PCA (principal component analysis). However, in this book, the choice of columns for `X` will be straightforward.

Fourth, non-numeric columns in `X` must be mapped to integer values. For example, the `sex` column consists of male and female values, which can be mapped to 0 and 1 via the `map()` method that is available in every Pandas data frame.

Fifth, different classification algorithms have different parameters, many of which have default values. This book does not delve into the details of classification algorithms to explain how they work or the purpose of their parameters. However, some parameters have an intuitive purpose, such as `n_neighbours` for the `kNN` algorithm, or `n_estimators` for the number of trees in the random forest algorithm, and so forth. However, the `criterion` parameter can be initialized with `entropy` or with `gini`, neither of which is discussed in this book. Search for online tutorials that provide details for the parameters of classification algorithms.

An Important Caveat

All the code samples for training models via classification algorithms select a subset of columns from a dataset. However, the selected columns do not necessarily generate the maximum accuracy. Although it is often straightforward to select the appropriate set of columns for datasets containing a handful of columns, the choice of columns can be non-intuitive and quite complex, especially for datasets that contain hundreds (or thousands) of columns.

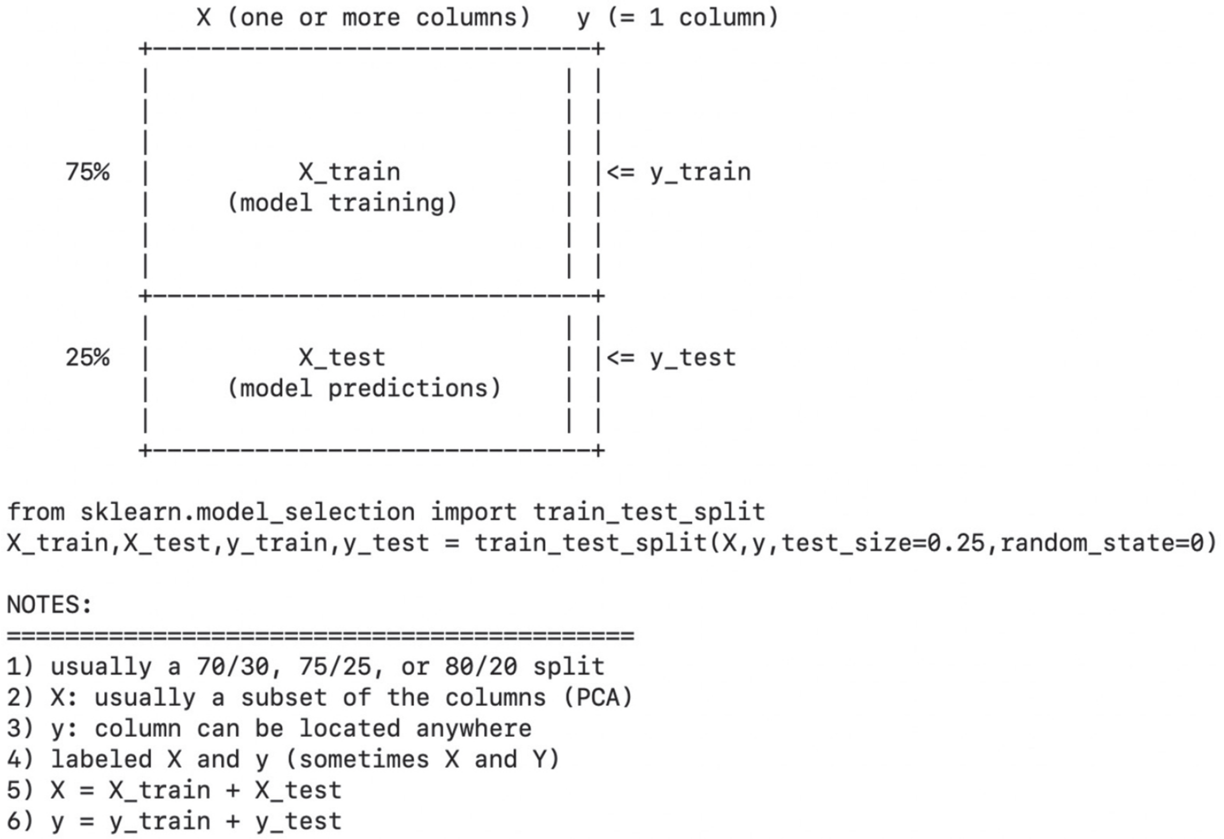
In fact, there are various techniques for programmatically determining the set of columns for training a model with a dataset. Such techniques include PCA, RFE (Recursive Feature Extraction), and many others. These techniques are outside the scope of this book. Instead, the primary purpose of the code samples in this chapter is to show you the sequence of steps that are required to train a classification-based model with a given dataset.

DIAGRAM FOR PARTITIONED DATASETS

This section contains a diagram that shows you the four subsets that are labeled `X_train, X_test, y_train`, and `y_test` that are created with the following code snippet:

```
X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.25,random_state=0
)
```

The preceding code snippet (or equivalent) appears in Python code samples for classification tasks. Figure 4.2 shows the relationship among the four subsets of data that are created via the scikit-learn function `train_test_split()`.



```
          X (one or more columns)    y (= 1 column)
        +-------------------------------+
        |                             | |
        |                             | |
        |                             | |
   75%  |          X_train            | |<= y_train
        |      (model training)       | |
        |                             | |
        |                             | |
        +-------------------------------+
        |                             | |
   25%  |          X_test             | |<= y_test
        |      (model predictions)    | |
        |                             | |
        +-------------------------------+

from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.25,random_state=0)

NOTES:
===========================================
1) usually a 70/30, 75/25, or 80/20 split
2) X: usually a subset of the columns (PCA)
3) y: column can be located anywhere
4) labeled X and y (sometimes X and Y)
5) X = X_train + X_test
6) y = y_train + y_test
```

FIGURE 4.2 A train test split diagram

The set `X` in Figure 4.2 is a subset of the data frame `df` that contains the contents of a CSV file, an example of which is shown here (and also shown in the code sample in the next section):

```
# populate the data frame df with four columns
from titanic2.csv:
df = pd.read_csv('titanic2.csv', usecols =
['survived','pclass','sex','age'])

# map male/female to the values 0 and 1:
df['sex'] = df['sex'].map({'male':0,'female':1})

from sklearn.model_selection import
train_test_split
```

```
# the target column y is the contents of
'survived':
y = df['survived']
df = df.drop(['survived'],axis = 1)

# the set X now consists of the columns
'pclass','sex','age':
X = df

# split into 75:25 ratio:
X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.25,random_state=0
)
```

A KNN-BASED MODEL WITH THE WINE.CSV DATASET

This section shows you how to perform the steps listed in a previous section to train a model with the kNN classification algorithm.

Listing 4.8 displays the content of `knn_wine.py` that shows you how to train a kNN-based model on the `wine.csv` dataset. Note that the comments containing a sequence of steps are the same steps that are listed in "Steps for Training Classifiers" in a previous section.

Listing 4.8: knn_wine.py

```
# Step 1: import required Python libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
# Step 2: populate a Pandas data frame df from a
CSV file
dataset = pd.read_csv('wine.csv')

# Step 3: specify columns for X and y (from df)
X = dataset.iloc[:, [0, 1]].values
y = dataset.iloc[:, 2].values

# Step 4: populate X_train, X_test, y_train, and
y_test
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size = 0.25,
random_state = 0)

# Step 5: perform feature scaling (X_train and
y_train)
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Step 6: create an instance classifier of a
specific classifier class
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 5,
metric='minkowski', p=2)

# Step 7: invoke the fit() method of classifier
classifier.fit(X_train, y_train)
```

```
# Step 8: invoke the predict() method of cls
y_pred = classifier.predict(X_test)

# Step 9: populate a confusion matrix cm
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Step 10: display the contents of the confusion
matrix cm
print("confusion matrix:")
print(cm)

# Step 11: inspect values for precision, recall,
and F1 score
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

Listing 4.8 starts with several `import` statements, and then the variable dataset is initialized with the contents of the CSV file `wine.csv`. The third step initializes the variable `X` with the first two features of the CSV file, and then initializes the variable `y` with the third feature of the CSV file.

The fourth step invokes the `train_test_split()` method in the same manner as you saw in a previous code sample in this chapter. The fifth step uses the `StandardScalar` class to scale the contents of `X_train` and `X_test` (note that the contents of `y_train` and `y_test` are already scaled).

The sixth step instantiates the variable `classifier` as an instance of the class `KNeighborsClassifier` that belongs to scikit-learn. The seventh step fits the model to the data, and the eighth step generates `y_pred`, which is a set of predictions for the data in `X_test`. The ninth step generates a confusion matrix based on the contents of `y_test` and `y_pred`. The tenth step prints the contents of the confusion matrix, and the eleventh step generates a report that contains values for precision, recall, and `f1` score. Launch the code in Listing 4.8, and you will see the following output:

```
confusion matrix:
[[15  0  1]
 [ 0 17  4]
 [ 0  1  7]]
             precision    recall  f1-score   support

           1       1.00      0.94      0.97        16
           2       0.94      0.81      0.87        21
           3       0.58      0.88      0.70         8

    accuracy                           0.87        45
   macro avg       0.84      0.87      0.85        45
weighted avg       0.90      0.87      0.88        45
```

OTHER MODELS WITH THE WINE.CSV DATASET

This section shows you how replace the kNN-specific code in the previous section with several other classifiers:

1. `DecisionTreeClassifier`
2. `RandomForestClassifier`
3. `GaussianNB`
4. `SVC`

For your convenience, here is the code block in Step 6 of Listing 4.18; it is the code that you need to replace with a different code block:

```
# Step 6: create an instance classifier of a
specific classifier class
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 5,
metric='minkowski', p=2)
```

In Listing 4.8 you need to specify different hyperparameters as well as the preceding code block if you specify the same dataset.

You must make additional code modifications if you specify a different dataset, which involves the following three steps:

- the name of the CSV file
- the set of columns in X
- the set of columns in y

Determining the columns of the dataset that are specified in the set `X` can be a complex task.

1) Here is the new code block for a decision tree (`DecisionTreeClassifier`):

```
from sklearn.tree import DecisionTreeClassifier
#classifier = DecisionTreeClassifier(criterion='entropy',random_state=0)
classifier =
DecisionTreeClassifier(criterion='gini',random_state=0)
classifier.fit(X_train, y_train)
```

2) Here is the new code block for a random forest (`RandomForestClassifier`):

```
from sklearn.ensemble import
RandomForestClassifier
classifier = RandomForestClassifier(n_estimators =
500, criterion='entropy', random_state = 0)
classifier.fit(X_train, y_train)
```

3) Here is the new code block for Naive Bayes (`GaussianNB`):

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

4) Here is the new code block for `SVC (SVC):`

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state =
0)
classifier.fit(X_train, y_train)
```

Notice that the preceding code blocks specify values for only one or two parameters: the other parameters for each algorithm have default values. You can perform a quick online search to find documentation regarding the parameters (and their default values) for the algorithms in the preceding code blocks. After reading their descriptions, decide which parameter values (if any) that you want to modify to experiment with the training process of the respective models. For example, the following webpage contains documentation for the parameters that are available for the decision tree algorithm:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Some algorithms provide parameters that are not intuitively obvious with respect to their purpose. For example, the `DecisionTreeClassifier` provide the criterion parameter whose value can be either `gini` or `entropy`, both of which require additional study to understand their purpose and how they differ from each other. A discussion of these parameter values is outside the scope of this book.

A KNN-BASED MODEL WITH THE BMI.CSV DATASET

Listing 4.9 displays a portion of the `bmi.csv` dataset, and Listing 4.20 displays the new code block that specifies the `bmi.csv` dataset and the appropriate columns for `X` and `y`.

Listing 4.9: bmi.csv

```
gender,age,height,safebmi
Male,19,190,0
Male,15,180,0
Female,16,150,0
// lines omitted for brevity
Female,16,130,0
Male,12,130,1
Female,13,150,1
Male,15,190,0
Male,16,170,1
```

Listing 4.20: knn_bmi.py

```
# Step 1: import required Python libraries
```

```
import pandas as pd
df = pd.read_csv('bmi.csv')

# map gender values Male/Female to 0/1:
df['gender'] =
df['gender'].map({'Male':'0','Female':'1'})

X = df.iloc[:, [0, 1, 2]].values
y = df.iloc[:, 3].values
// lines omitted for brevity
```

Listing 4.9 contains the initial portion of `knn-bmi.py`, and the remaining code is the same as `knn_wine.py`. Launch the code in Listing 4.9, and you will see the following output:

```
confusion matrix:
[[39 12]
 [18  6]]
classification report:
              precision    recall  f1-score   support

           0       0.68      0.76      0.72        51
           1       0.33      0.25      0.29        24

    accuracy                           0.60        75
   macro avg       0.51      0.51      0.50        75
weighted avg       0.57      0.60      0.58        75
```

A KNN-BASED MODEL WITH THE DIABETES.CSV DATASET

Listing 4.10 displays a portion of the `diabetes.csv` dataset, and Listing 4.11 displays the new code block for `knn_diabetes.py` that specifies the `diabetes.csv` dataset and the appropriate columns for `X` and `y`.

Listing 4.10: diabetes.csv

```
Pregnancies,Glucose,BloodPressure,SkinThickness,In
sulin,BMI,DiabetesPedigreeFunction,Age,Outcome
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
// lines omitted for brevity
2,122,70,27,0,36.8,0.34,27,0
5,121,72,23,112,26.2,0.245,30,0
1,126,60,0,0,30.1,0.349,47,1
1,93,70,31,0,30.4,0.315,23,0
```

Listing 4.11: knn_diabetes.py

```
# Step 1: import required Python libraries
import pandas as pd
df = pd.read_csv('diabetes.csv')

# Step 3: specify columns for X and y (from df)
*** first set ***
# BloodPressure,BMI,Age:
X = dataset.iloc[:, [2, 5, 7]].values
# Outcome:
y = dataset.iloc[:, 8].values
// lines omitted for brevity
```

Listing 4.11 contains the initial portion of `knn_diabetes.py` that shows you the new code block in bold. The remaining code is the same as `knn_bmi.py`. Launch the code in Listing 4.11, and you will see the following output:

```
confusion matrix:
[[39 12]
 [18  6]]
classification report:
              precision    recall  f1-score   support

           0       0.68      0.76      0.72        51
           1       0.33      0.25      0.29        24

    accuracy                           0.60        75
   macro avg       0.51      0.51      0.50        75
weighted avg       0.57      0.60      0.58        75
```

Replace the code shown in bold in Listing 4.11 with the following code block:

```
# Step 3: specify columns for X and y (from df)
# all columns:
X = dataset.iloc[:, [0, 1, 2, 3, 4, 5, 6,
7]].values
# Outcome:
y = dataset.iloc[:, 8].values
```

Launch the code in Listing 4.11, and you will see the following output:

```
confusion matrix:
[[114  16]
 [ 22  40]]
              precision    recall  f1-score   support

           0       0.84      0.88      0.86       130
           1       0.71      0.65      0.68        62

    accuracy                           0.80       192
   macro avg       0.78      0.76      0.77       192
weighted avg       0.80      0.80      0.80       192
```

As you can see, the `precision` and `recall` have increased (i.e., improved) in the second set, which contains eight columns in `diabetes.csv`, whereas the first set contains three columns in `diabetes.csv`.

SMOTE AND THE TITANIC DATASET

This section is optional because there are various code-related details that have not been discussed thus far, which are listed here:

- Creating a Pandas data frame from a CSV file
- Replacing categorical data with numeric values
- Invoking `train_test_split()`
- Invoking the `fit()` and `test()` methods
- Precision and recall

However, everything in the preceding bullet list is covered in an introductory machine learning course, which is a recommended (but not mandatory) prerequisite for this book.

Listing 4.12 displays the content of `smote_titanic.py` that illustrates how to concatenate a set of CSV files into a single CSV file using a Pandas data frame.

Listing 4.12: smote_titanic.py

```
import pandas as pd
import numpy as np

# pip3 install imblearn <= if not installed
# import SMOTE module from imblearn library
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import
LogisticRegression
from sklearn.metrics import confusion_matrix,
classification_report

df = pd.read_csv('titanic2.csv', usecols =
['survived','pclass','sex','age'])
df['sex'] = df['sex'].map({'male':0,'female':1})
print(df)

from sklearn.model_selection import
train_test_split

y = df['survived']
df = df.drop(['survived'],axis = 1)
X = df

# split into 75:25 ratio:
X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.25,random_state=0
)
```

```
# create logistic regression object, then fit,
then predict:
lr1 = LogisticRegression()
lr1.fit(X_train, y_train.ravel())
predictions = lr1.predict(X_test)

cm = confusion_matrix(y_test, predictions)
print()
print("=> confusion matrix:")
print(cm)
print()

print(classification_report(y_test, predictions))

print("=> Before OverSampling:")
print("label '1': {}".format(sum(y_train == 1)))
print("label '0': {}".format(sum(y_train == 0)))
print()

# perform oversampling with SMOTE:
sm = SMOTE(random_state = 2)
X_train_res, y_train_res = sm.fit_sample(X_train,
y_train.ravel())

print("=> After OverSampling:")
print("label '1': {}".format(sum(y_train_res ==
1)))
print("label '0': {}".format(sum(y_train_res ==
0)))
print()
```

```
# create logistic regression object, then fit,
then predict:
lr2 = LogisticRegression()
lr2.fit(X_train_res, y_train_res.ravel())
predictions = lr2.predict(X_test)

cm = confusion_matrix(y_test, predictions)
print("=> confusion matrix:")
print(cm)
print()

# print classification report
print(classification_report(y_test, predictions))
```

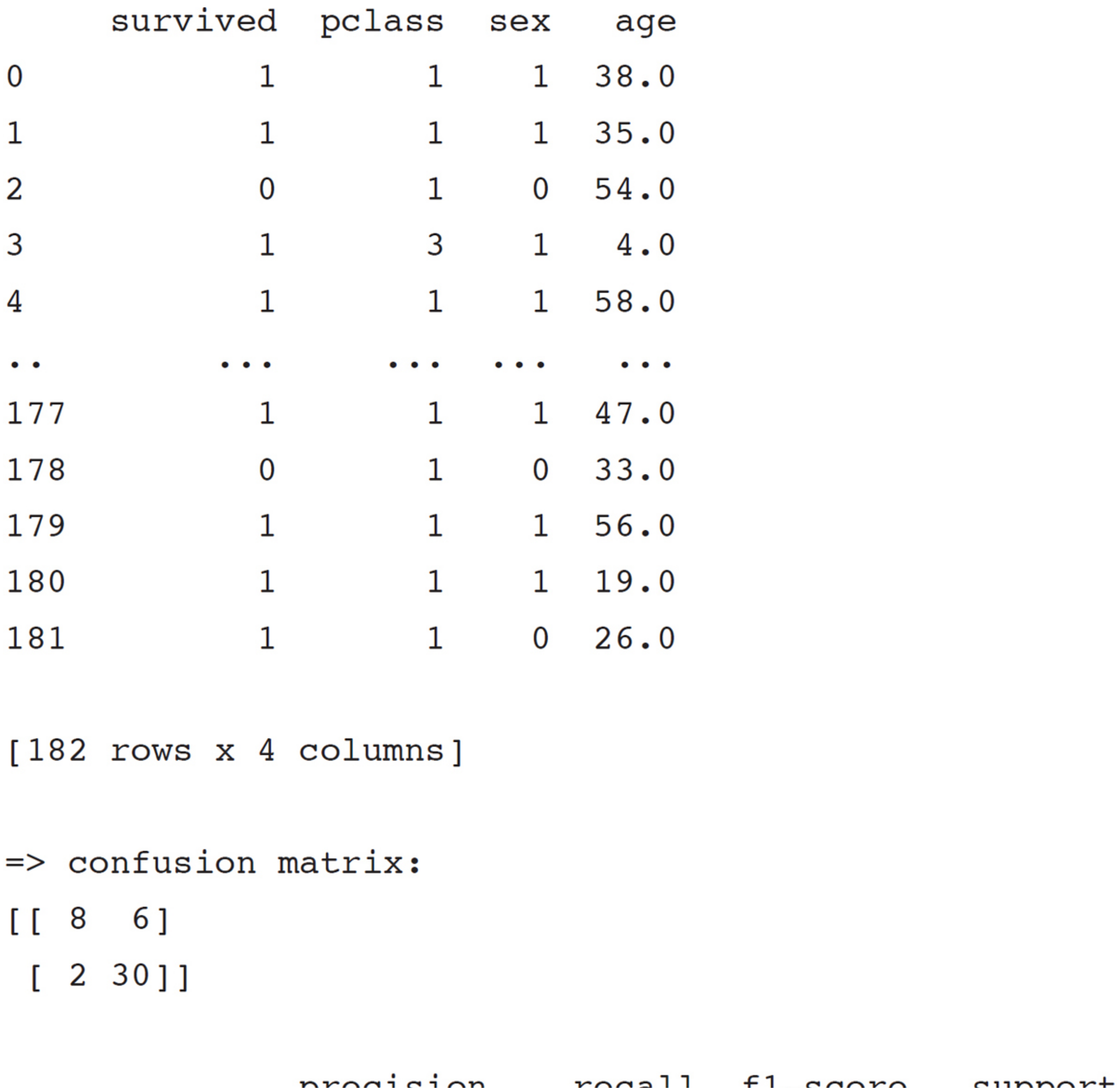
Listing 4.12 starts with several `import` statements, followed by the initialization of the Pandas data frame `df` with four columns from the CSV file `titanic2.csv`. Next, the data in the `sex` column is replaced with values 0 and 1 for male and female, respectively.

The next code block initializes `y` with the survived column, drops this column from the data frame `df`, and initializes `X` with the modified data frame. At this point, we can invoke the method `train_test_split()` to initialize `X_train`, `X_test`, `y_train`, and `y_test.`

The next portion of Listing 4.12 instantiates `lr1` as an instance of the Python-based `LogisticRegression` class, fits the training data on this model, makes a prediction on the test data, and then displays the confusion matrix as well as the classification report. The next snippet displays the class distribution based on the current contents of the data frame `df`, followed by two lines of code that perform the oversampling on the `df`, as shown here:

```
sm = SMOTE(random_state = 2)
X_train_res, y_train_res = sm.fit_sample(X_train,
y_train.ravel())
```

The class distribution is displayed again, which shows that we now have a balanced class in the data frame `df`. The final code block performs a similar set of operations on the variable `lr2` that was performed on the variable `lr1`. Launch the code in [Listing 4.12](#), and you will see the following output:



```
     survived  pclass  sex   age
0           1       1    1  38.0
1           1       1    1  35.0
2           0       1    0  54.0
3           1       3    1   4.0
4           1       1    1  58.0
..        ...     ...  ...   ...
177         1       1    1  47.0
178         0       1    0  33.0
179         1       1    1  56.0
180         1       1    1  19.0
181         1       1    0  26.0

[182 rows x 4 columns]

=> confusion matrix:
[[ 8  6]
 [ 2 30]]

                precision    recall  f1-score   support
```

```
              precision    recall  f1-score   support

           0       0.80      0.57      0.67        14
           1       0.83      0.94      0.88        32

    accuracy                           0.83        46
   macro avg       0.82      0.75      0.77        46
weighted avg       0.82      0.83      0.82        46

=> Before OverSampling:
label '1': 91
label '0': 45
=> After OverSampling:
label '1': 91
label '0': 91

=> confusion matrix:
[[13  1]
 [ 6 26]]

              precision    recall  f1-score   support

           0       0.68      0.93      0.79        14
           1       0.96      0.81      0.88        32

    accuracy                           0.85        46
   macro avg       0.82      0.87      0.83        46
weighted avg       0.88      0.85      0.85        46
```

Compare the two confusion matrices and notice that `TP` has noticeably increased after the oversampling has been performed, with a flip of the values for `FN` and `FP`. In addition, notice that the class distribution is displayed again, which shows you that we now have a balanced class. Last, notice that the recall value has increased in label 0, which was initially an under-represented label in the dataset.

EDA AND DATA VISUALIZATION

Let’s take a brief look at EDA and histograms, as well as EDA and heat maps, both of which are discussed in the following subsections.

EDA and Histograms

One technique for analyzing the distribution of the data in a dataset is to render the data in a histogram. Listing 4.13 displays the content of `hist_titanic.py` that shows you how to render the data in the `Titanic` dataset in a histogram.

Listing 4.13: hist_titanic.py

```
import pandas as pd
import matplotlib.pyplot as plt

filename="titanic2.csv"
train_df = pd.read_csv(filename)
train_df2 = train_df[['survived','age']]
train_df2['age'] = train_df2['age'].astype(int)

train_df2.hist("age",by="survived",edgecolor='blu
e',linewidth=1,grid=True,color="red",figsize=(10,
8),bins=30)
plt.suptitle("")
plt.xlabel("")
plt.show()
```

Listing 4.13 starts with two `import` statements and then initializes the Pandas data frame `train_df` with the contents of the CSV file `titanic.csv`. The next code snippet initializes the Pandas data frame `train_df2` with the values of the features `survived` and `age`. Next, the contents of `train_df2['age']` are treated as integer-based values.

The next code snippet invokes the `hist()` method that is part of the `train_df2` data frame to render a histogram whose axes are the `age` and `survived` features, followed by snippets for the title and label for the horizontal axis (both are empty strings). Launch the code in Listing 4.13, and you will see the histogram that is shown in Figure 4.3.

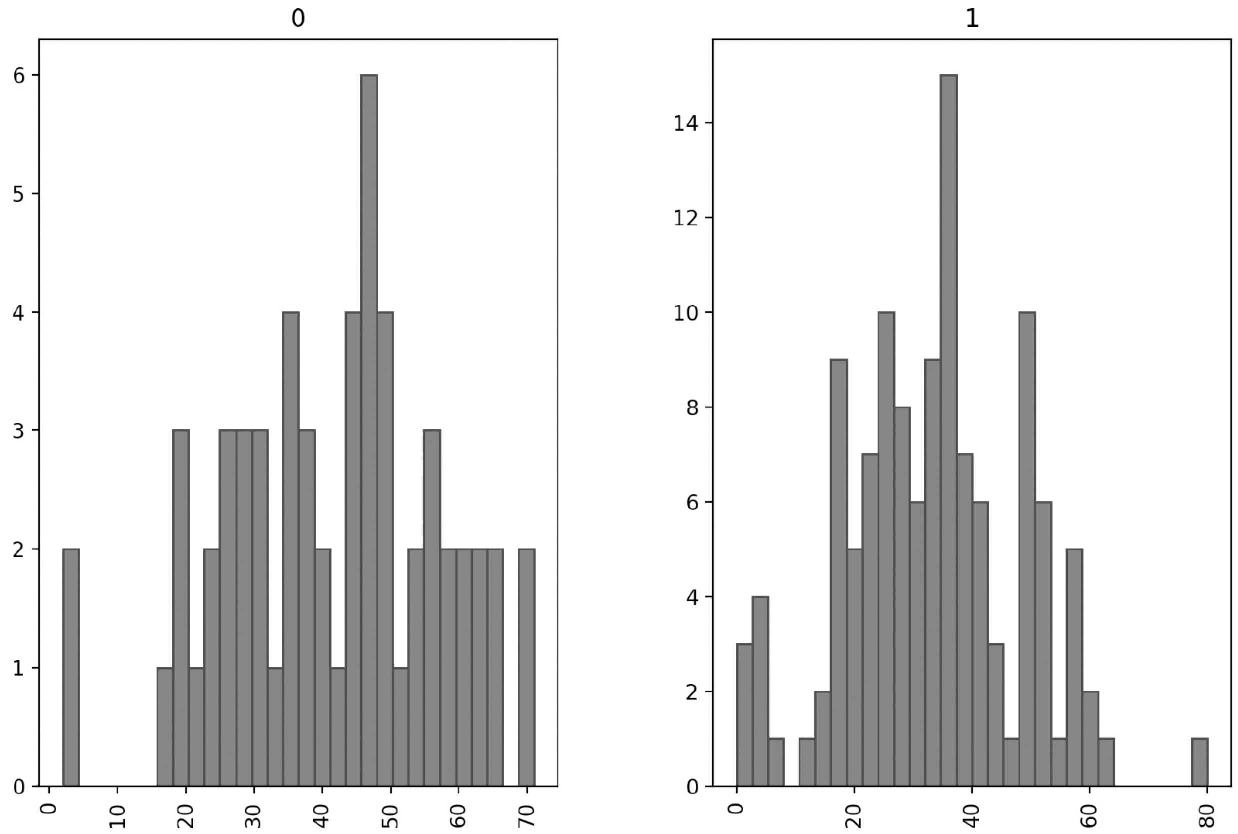


FIGURE 4.3 A histogram with data from the Titanic dataset

EDA and Heatmaps

One technique for examining the correlation between features in a dataset involves a heat map. Listing 4.14 displays the content of `heatmap_titanic.py` that shows you how to render the data in the `Titanic` dataset in a histogram.

Listing 4.14: heatmap_titanic.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
filename="titanic2.csv"
df = pd.read_csv(filename)
matrix = df.corr()
plt.figure(figsize=(8,6))
sns.heatmap(matrix,cmap='Blues',annot=True)
plt.show()
```

Listing 4.14 starts with four `import` statements and then initializes the Pandas data frame `df` with the contents of the CSV file `titanic.csv`. The next code snippet initializes the variable `matrix` as the correlation matrix for the features in `df`.

After specifying the dimensions of the output figure, the Seaborn `heatmap()` is invoked to render a heat map. Launch the code in Listing 4.14, and you will see the histogram that is shown in Figure 4.4.

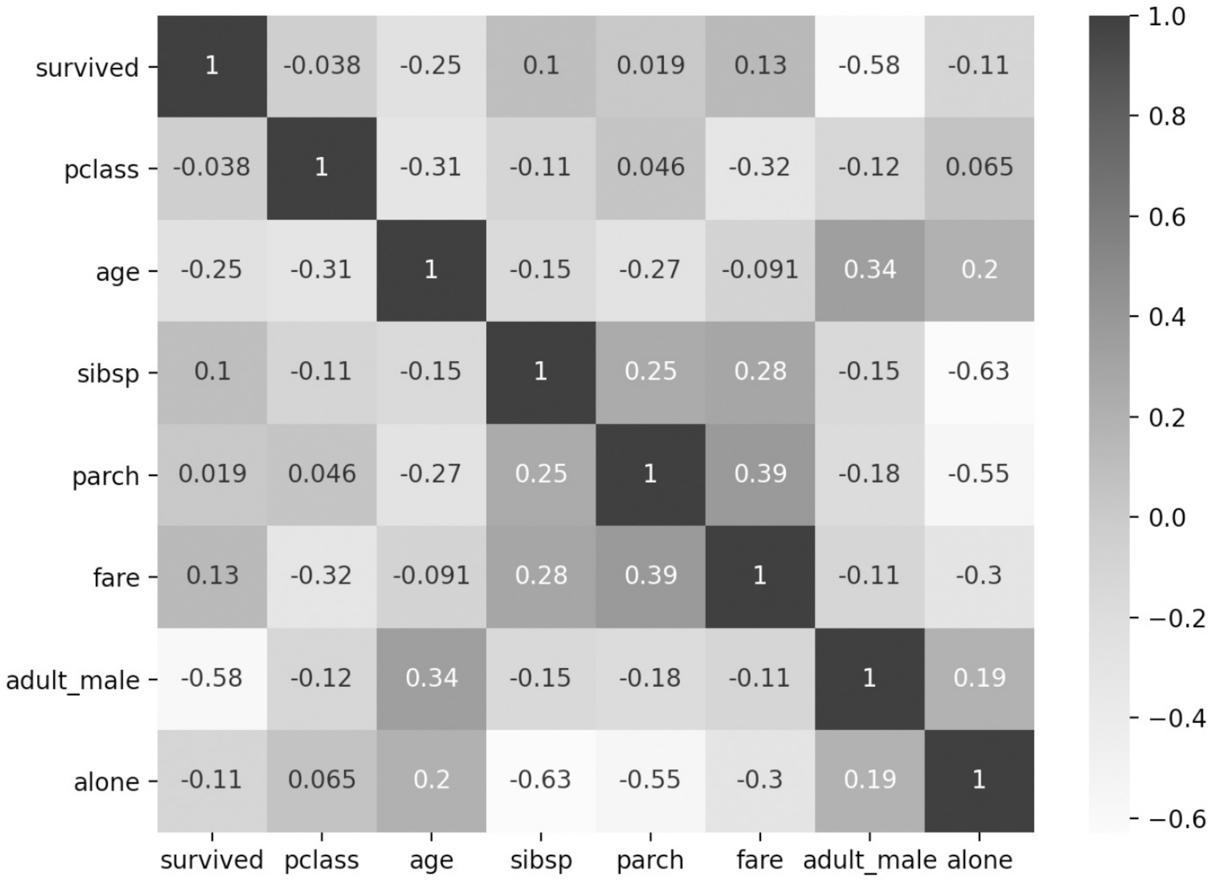


FIGURE 4.4 A heat map with data from the Titanic dataset

WHAT ABOUT REGRESSION AND CLUSTERING?

Regression (which includes linear regression) is another type of supervised learning task with datasets that have a target column that can contain a wide range of numeric values, whereas classification algorithms have a target column with a much smaller number of numeric values.

Clustering is an unsupervised learning task in which datasets do *not* have a target column. A dozen or so algorithms exist for clustering, including kMeans (probably the most well-known algorithm) and meanShift.

Regression and clustering examples are not included because the focus of this book is for classification algorithms in machine learning. However, you can perform an online search, and you will find numerous free tutorials and blogposts that provide examples of linear regression and clustering.

FEATURE IMPORTANCE

In general, you want to determine which features in a dataset are significant and their relative importance. For example, the number of bedrooms and bathrooms in a house are significant features that affect the price of a house, whereas the ticket number for an airplane ticket is most likely not a significant feature.

Suppose that a dataset has a “target” column, which is a column whose values we want to predict during the testing phase (i.e., after the training phase). A target column exists in datasets for classification problems (which includes linear regression and classification algorithms), whereas datasets without a target column are clustering tasks.

The key idea pertaining to feature importance involves assigning a numeric score between 0 and 100 that is assigned to each feature, which measures the extent to which that feature can predict the target variable. The conditions for assigning feature importance are summarized here:

- Select all the relevant features.
- Assign a value between 0 and 100 to each feature.
- Ensure the sum of the feature values is 100.

Note: If we divide the feature values by 100, we have a probability distribution.

Decision Trees and Feature Importance

One way to determine feature importance involves the random forest algorithm in scikit-learn. For a given feature, scikit-learn calculates its importance (which is a numeric value) by examining the average reduction in the impurity in the set of tree nodes that involve that feature. This calculation is performed for each feature after the training step has completed, after which the values are scaled (essentially a probability distribution) to simplify the comparison of the importance of features.

Listing 4.15 displays the content of `feature_importance.py` that shows you how to determine the important features in the `Titanic` dataset, as well as their relative importance.

Listing 4.15: feature_importance.py

```
from scikit-learn.model_selection import
train_test_split
from scikit-learn.ensemble import
RandomForestClassifier

import pandas as pd
import numpy as np

df = pd.read_csv("titanic2.csv")
df =
df[["survived","sex","age","class","embarked"]]
y = df.pop("survived")

# convert categories to numeric values:
df['sex'] = df['sex'].replace({'male':1,
"female":0})
df['class'] = df['class'].replace({'First':1,
"Second":2, "Third":3})
```

```
df['embarked'] = df['embarked'].replace({'S':0,
"C":1, "Q":2})

# perform train test split:
X_train, X_test, y_train, y_test =
train_test_split(df, y, test_size=0.2,
random_state=42)
rng = np.random.RandomState(0)
random_forest =
RandomForestClassifier(n_estimators=10,
random_state=rng)
random_forest.fit(X_train,y_train)

# determine the relative feature importance:
feature_imp =
pd.DataFrame({'feature':X_train.columns,'importanc
e':
np.round(random_forest.feature_importances_,3)})

feature_imp = feature_imp.sort_values('importanc
e',ascending=False).set_index('feature')
print("feature_imp.head():")
print(feature_imp.head())
```

Listing 4.15 starts with four `import` statements and then initializes the Pandas data frame `df` with the contents of the CSV file `titanic2.csv`. Next, the variable `df` is reduced to five attributes, after which the variable `y` is initialized with the contents of the `survived` feature, as shown here:

```
df =
df[["survived","sex","age","class","embarked"]]
y = df.pop("survived")
```

The next block of code involves three invocations of the `replace()` method to replace the values of the `sex`, `class`, and `embarked` features with a corresponding set of numeric values.

Now we can split the contents of `df` and `y` into four sub-regions in preparation for the training step and the test step, as shown in this code snippet:

```
X_train, X_test, y_train, y_test =
train_test_split(df, y, test_size=0.2,
random_state=42)
```

The next portion of Listing 4.15 initializes the variable `random_forest` as an instance of the `RandomForestClassifier` class, after which the `fit()` method is invoked to train the model.

The final portion of Listing 4.15 accesses the relative importance of the features and displays their values as a decimal number (note that their sum equals 1). Launch the code in Listing 4.15, and you will see the following output:

```
feature_imp.head():
         importanc
                 e
feature
age          0.583
sex          0.346
class        0.038
embarke      0.033
d
```

WHAT IS FEATURE ENGINEERING?

Feature engineering is the process of determining a new set of features that are based on a combination of existing features to create a meaningful dataset for a given task. Domain expertise is often required for this process, even in cases of relatively simple datasets. Feature engineering can be tedious and expensive, and in some cases, you might consider using automated feature learning. After you have created a dataset, it is a good idea to perform feature selection or feature extraction (or both) to ensure that you have a high quality dataset.

After creating a dataset and cleaning its values, examine the features in the dataset: are there features that are clearly important? If so, then you can perform features section by selecting those features. Visual inspection does not guarantee that you can determine the complete set of significant features, nor can you guarantee the relative importance of those selected features.

Another approach is *feature extraction*, which involves programmatically determining the most relevant features in the dataset. Some feature extraction techniques calculate linear combinations of existing features (e.g., PCA), whereas other techniques involve non-linear combinations (such as t-sne).

In addition to performing feature selection or feature extraction, you can also perform feature importance to determine the relative importance of features in a dataset.

In high-level terms, some algorithms adopt a “bottom-up” approach whereby an initial set of relevant features starts with a single feature. Each time that another feature is added, the predictive accuracy of the feature set is calculated: features that increase the accuracy are maintained in the feature set, and those that do not increase the accuracy are discarded.

However, some algorithms adopt a “top-down” approach that starts with a larger initial set of features. A feature is chosen for removal from the initial set of features, and then the accuracy of the reduced set of features is calculated: if the accuracy decreases, then the removed feature is returned to the initial dataset, and if the accuracy increases, then another feature is chosen for removal, and the process is repeated.

WHAT IS FEATURE SELECTION?

Feature selection is also called *variable selection*, *attribute selection*, or *variable subset selection*. Feature selection involves selecting a subset of the most relevant features in a dataset, which provides these advantages:

- reduced training time
- simpler models that are easier to interpret
- avoidance of the curse of dimensionality
- better generalization due to reduced overfitting (“reduction of variance”)

Feature selection techniques are often used in domains where there are many features and comparatively few samples (or data points). A low-value feature can be redundant or irrelevant, which are two different concepts. For instance, a relevant feature might be redundant when it is combined with another strongly correlated feature.

Sometimes datasets contain a pair of features in which the categorical values of feature `A` are essentially a subset of the values in feature `B`. Determine whether there is no loss (or an acceptable loss) of information as a result of combining feature `A` and feature `B` into a single feature.

Another simple example involves splitting a feature into multiple features. For example, suppose that feature `A` contains the first name and last name of a set of customers. If you need to process that data based on a customer's last name, consider splitting feature `A` into two features (first name and last name). However, you might already have a pair of features for the first name and last name of customers and merging them into one feature might be preferable (depends on the specific use-case).

Feature selection can be employed for performing regression tasks as well as classification tasks. Supervised feature selection techniques have the following properties:

- They take into account the target variable.
- Some remove irrelevant variables.
- Some use a filter strategy (e.g., information gain).
- Some use a wrapper strategy (e.g., search guided by accuracy).
- Some use the embedded strategy.

In the embedded strategy, prediction errors are used to determine whether features are included or excluded while developing a model.

An example of a filter-based algorithm is XGBoost. Examples of a wrapper-based algorithm include GA as well as the RFE feature selection technique. An example of an embedded-based algorithm is the L1 Lasso method.

A more recent technique for determining dependencies that works with numeric and categorical data is discussed online at

https://phik.readthedocs.io/en/latest/#

Classification of Feature Selection Techniques

Several types of feature selection techniques are shown here:

- filter methods
- wrapper methods
- embedded methods

- hybrid methods

In brief, *filter methods* for feature selection include methods that use the distributions of features to determine the selection of features. Such techniques are not as effective for selecting features, even though they execute very quickly. Examples of filter methods include chi-squared and ANOVA (Analysis of Variance).

These methods are computationally very fast, but in practice, they do not render good features for our models. In addition, when we have big datasets, p-values for statistical tests tend to be very small, highlighting as significant tiny differences in distributions that may not be really important.

Wrapper methods include greedy algorithms, which can be infeasible due to their exhaustive examination of possible combinations of features in a dataset. These methods use an iterative approach: select an optimal set of features, evaluate the performance, and then repeat this process. Examples of wrapper methods include cross-validation, forward selection, and RFE, all of which are discussed in online articles.

Embedded methods for feature selection leverage the advantages of filter methods as well as wrapper methods. Embedded methods first train a machine learning model, after which they use the trained model's feature importance to select features. Although the techniques in this group also execute quickly, we can only use the algorithms that provide feature importance. *Hybrid methods* involve combinations of the other types of methods to perform feature selection, which includes the following:

- feature shuffling
- feature performance
- target mean performance

Navigate to the following Web page (specifically see diagram #4) for filter methods, embedded methods, wrapper methods, and hybrid methods for feature selection:

https://towardsdatascience.com/feature-selection-for-the-lazy-data-scientist-c31ba9b4ee66

Feature Selection Algorithms

Before we look at any algorithms, keep in mind that machine learning algorithms such as Lasso, decision trees, and random forests automatically perform feature selection during the training step of a given model.

By contrast, *unsupervised* feature selection techniques do not involve a target feature because a target feature does not exist in the dataset. Instead, some of these selection techniques remove redundant variables via correlation. Here is a list of relevant feature selection techniques:

- Backward Feature Elimination
- Factor Analysis
- Forward Feature Selection
- Independent Component Analysis
- LOCO (Leave One Covariate Out)
- RFE

For example, Listing 4.16 displays the content of `rfe1.py` that shows you how to use `RFE` to perform feature extraction.

Listing 4.16: rfe1.py

```
from scikit-learn.datasets import
make_classification
from scikit-learn.feature_selection import RFE
from scikit-learn.tree import
DecisionTreeClassifier
```

```
# define data set
X, y = make_classification(n_samples=1000,
n_features=10, n_informative=5, n_redundant=5,
random_state=42)

# define and then fit RFE:
rfe = RFE(estimator=DecisionTreeClassifier(),
n_features_to_select=5)
rfe.fit(X, y)

# summary of selected/not-selected features:
for i in range(X.shape[1]):
  print(f'Column: {i} Chosen: {rfe.support_[i]:4}
Rank: {rfe.ranking_[i]:4}')
```

Listing 4.16 starts with several `import` statements and then initializes the variables `X` and `y` by invoking the `make_classification()` method that generates random values. In this case, there are 1000 rows and 10 features.

The next portion of Listing 4.16 invokes the `RFE()` method with an instance of the class `DecisionTreeClassifier` as the value for the estimator parameter, and also specifies that 5 features need to be selected. Launch the code in Listing 4.16, and you will see the following output:

```
Column: 0 Selected:    0 Rank:    6
Column: 1 Selected:    1 Rank:    1
Column: 2 Selected:    0 Rank:    2
Column: 3 Selected:    1 Rank:    1
Column: 4 Selected:    1 Rank:    1
Column: 5 Selected:    1 Rank:    1
Column: 6 Selected:    0 Rank:    4
Column: 7 Selected:    1 Rank:    1
Column: 8 Selected:    0 Rank:    5
Column: 9 Selected:    0 Rank:    3
```

Continuous Versus Categorical Features

Categorical features must be mapped to numeric values, and one well-known technique for doing so is called *one-hot encoding*. For instance, if a categorical feature consists of the three colors red, green, and blue, you can represent each color by a 1x3 vector, as shown here:

- [1,0,0] for red
- [0,1,0] for green
- [0,0,1] for blue

In general, if a categorical feature consists of n distinct values, you can represent each of these values as a 1xn vector from the nxn identity matrix. However, feature selection (and feature extraction) algorithms are unaware of the “binding” that exists among the one-hot encoded columns.

Determining a Feature Selection Algorithm

Earlier, you saw a list of machine learning algorithms you can use for feature selection. This section recommends specific algorithms to perform feature selection.

If you have a supervised learning task, try decision trees (or random forest) because they perform feature selection as part of their algorithm. If you want to try other algorithms after working with a decision tree or random forest, try RFE or a feature importance method. If you have an unsupervised learning task, start with `kMeans`, `meanShift`, or `meanShift++`.

WHAT IS FEATURE EXTRACTION?

Feature extraction creates new features from functions that produce combinations of the original features. By contrast, feature selection involves determining a subset of the existing features. The net effect of feature selection and feature extraction results in *dimensionality reduction* for a given dataset, which is not discussed in this book.

By contrast, feature exclusion involves retaining features that might be relevant for predicting the output. Moreover, feature exclusion involves dropping-vs-keeping features for training a model, whereas feature extraction involves deriving new (and ideally fewer) features from the existing features.

In some cases, you can perform a visual inspection of the features of a dataset to determine the most important features. However, visual inspection is not as reliable for larger datasets that contain dozens or hundreds of features. Fortunately, there are algorithms that can determine the most important features of a dataset, some of which are listed in the next section.

Feature Extraction Algorithms

There are powerful algorithms that perform feature extraction, and they typically involve non-trivial mathematical concepts. For example, PCA involves calculating eigenvalues and eigenvectors (standard material for mathematics majors). As such, this section contains a short list of algorithms, and you can perform an online search if you want to learn more about the details of these algorithms:

- PCA
- Independent Component Analysis (ICA)
- Linear Discriminant Analysis (LDA)

Note that the algorithms in the preceding list perform linear extraction, whereas the following list contains non-linear extraction algorithms:

- Locally Linear Embedding (LLE)
- t-distributed Stochastic Neighbor Embedding (t-SNE)
- Auto encoders

LLE is a dimensionality reduction technique that involves "manifold learning," and you can learn more details about this concept online:

https://cs.nyu.edu/~roweis/lle/

The t-sne algorithm involves the Kullback-Leibler (KL) divergence, which is a non-trivial concept. A suggestion: if you are unfamiliar with KL divergence, first learn about entropy and cross entropy, after which it will be easier for you to understand KL divergence.

Autoencoders are neural networks whose input layer and output layer are identical, and more details are available online:

https://en.wikipedia.org/wiki/Autoencoder

You can also find many online tutorials that contain code samples that illustrate how to use dimensionality reduction algorithms, such as the following:

https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be

Yet another useful technique is called *feature hashing* (distantly analogous to the “kernel trick” in SVMs), which is discussed here:

https://dzone.com/articles/feature-hashing-for-scalable-machine-learning

DATA CLEANING AND MACHINE LEARNING

Data cleaning can be an enormously time-consuming task that involves a significant amount of effort and domain knowledge. After you have cleaned a dataset, you need to decide which machine learning algorithms to employ for training a model on the dataset. A labeled dataset enables you to choose from various algorithms, including:

- logistic regression
- Naive Bayes
- decision trees
- random forests
- SVMs
- CNNs (part of deep learning)

If you have unlabeled data, then you can explore one or more of the following clustering algorithms:

- kMeans
- meanShift
- kMeans++
- kMedioid
- DBScan

Another consideration involves *imbalanced datasets*, which can adversely affect the accuracy of some machine learning algorithms. There are techniques for handling imbalanced datasets, which are discussed in Chapter 2.

Labeled Versus Unlabeled Data

Labeled data is often preferred because it saves you from performing the task of labeling the data yourself. However, most data is unlabeled, and the cost of labeling that data can be prohibitive. While it is straightforward and inexpensive to manually label pictures of dogs and cats, it is much more costly to label x-rays that contain signs of tumors.

Synthetic Data Labels

Synthetic labeling generates training data that resembles real data, which can be done in several ways, some of which are listed here:

- GANs (Generative Adversarial Networks)
- ARs (AutoRegressive models)
- VAEs (Variational Autoencoders)
- SSL (self-supervised learning)

The techniques in the preceding list are outside the scope of this book, but you can perform an Internet search and find numerous articles and blog posts. Another option involves commercial products from various companies that you can explore through an online search to determine which ones meet your needs.

Keep in mind that generating synthetic *labels* is different from generating *synthetic data*: examples of the latter (such as SMOTE) are discussed in Chapter 3.

Training Large Datasets

If you have a dataset that is over 100 MB (or whatever you consider to be large) and you want to obtain results more quickly, the following suggestions might save you a significant amount of time.

First, perform data cleaning your steps on a 10% subset of the data. If 10% is still too large, then reduce that number to 5% or even a 1% subset of the dataset. Second, use the smaller dataset in the previous step with each of the algorithms that you plan to use. This approach will give you faster results than the time for training the model just once on the entire dataset.

After completing the preceding sequence of steps, try randomly selecting yet another 10% (or less) and randomly select one of the algorithms to train the model and see if the results are comparable.

Finally, train the model on the entire dataset with all the algorithms and ideally the results will be comparable. However, if there are differences then you could explore the data to determine if and where there might be something anomalous in the data.

Of course, avail yourself of the GPU in Google Colaboratory (or some other cloud-based service) because it is probably much faster and also free to use for at least 10 hours on a daily basis.

Other Data-Related Topics

Although this chapter is oriented toward data quality, there are other concerns pertaining to data that are not covered in this chapter. Even if you are not directly involved, it is a good idea to be aware of potential concerns, some of which are shown here:

- data privacy
- data protection
- data security

Various techniques are available for the data-related topics in the previous list, some of which are listed here:

- cryptography
- data deletion
- encryption

If need be, you can perform an online search for more information about these techniques and data-related topics.

SUMMARY

This chapter started with techniques for scaling data and the importance of performing this task. Next, you learned how to scale numeric data via normalization, standardization, and via units of measure.

In addition, you learned about the confusion matrix as well as metrics such as precision, recall, specificity, accuracy, and F1 score. Moreover, you learned about the ROC curve and AUC curve for evaluating a trained model.

Then you saw how to train a model using the kNN algorithm with various datasets, including a wine dataset, a BMI dataset, and a diabetes dataset. You also learned about the SMOTE algorithm for generating synthetic data.

Moreover, you learned about feature engineering, which comprises feature selection and feature extraction. Finally, you learned about data cleaning, which can involve working with labeled and unlabeled data.

OceanofPDF.com

CHAPTER 5

MATPLOTLIB AND SEABORN

This chapter introduces data visualization, along with a collection of Python-based code samples that use Matplotlib to render charts and graphs. In addition, this chapter contains visualization code samples that combine Pandas and Matplotlib.

The first part of this chapter briefly discusses data visualization, with a short list of some data visualization tools, and a list of various types of visualization (bar graphs, pie charts, and so forth). There is a very short introduction to Matplotlib, followed by code samples that display the available styles in colors in Matplotlib.

The second part of this chapter contains an assortment of Python code samples that render horizontal lines, slanted lines, and parallel lines. This section also contains a set of code samples that show you how to render a grid of points in several ways.

The third part of this chapter shows you how to load images, display a checkerboard pattern, and plot trigonometric functions in Matplotlib. The fourth section contains examples of rendering charts and graphs in Matplotlib, which includes histograms, bar charts, pie charts, and heat maps.

The fifth section contains code samples for rendering 3D charts, financial data, and data from a sqlite3 database.

This chapter introduces several tools for data visualization, including Seaborn, Bokeh, and YellowBrick, along with an introduction to scikit-learn.

You will get an introduction to Seaborn for data visualization, which is a layer above Matplotlib. Although Seaborn does not have all of the features that are available in Matplotlib, Seaborn provides an easier set of APIs for rendering charts and graphs.

Also included here is a very short introduction to Bokeh, along with a code sample that illustrates how to create more artistic graphics effects with relative ease.

Finally, we delve into scikit-learn, which is a very powerful Python library that supports many machine learning algorithms and visualization. If you are new to machine learning, fear not: *this section does not require a background in machine learning to understand the Python code samples.*

IMPORT STATEMENTS FOR THIS CHAPTER

The following list contains all the `import` statements that you will encounter in the Python code samples for this chapter:

- `from bokeh.layouts import column`
- `from bokeh.plotting import figure, output_file, show`
- `from datetime import datetime`
- `from itertools import product`
- `from matplotlib import colors`

- `from matplotlib import pyplot as plt`
- `from matplotlib import style`
- `from sklearn import datasets`
- `from sklearn.datasets import load_digits`
- `from sklearn.model_selection import train_test_split`
- `from sklearn.preprocessing import StandardScaler`
- `import bokeh.colors as colors`
- `import math`
- `import matplotlib`
- `import matplotlib.pyplot as plt`
- `import mplfinance as mpf`
- `import numpy as np`
- `import pandas as pd`
- `import pylab`
- `import random`
- `import seaborn as sns`
- `import sqlite3`
- `import sweetviz as sv`
- `import sys`

WHAT IS DATA VISUALIZATION?

Data visualization refers to presenting data in a graphical manner, such as bar charts, line graphs, and heat maps. As you probably know, big data comprises massive amounts of data and leverages data visualization tools to assist in making better decisions.

Good data visualization tells a meaningful story, which in turn focuses on useful information that resides in datasets that can contain many data points (i.e., billions of rows of data). Another aspect of data visualization is its effectiveness: how well does it convey the trends that might exist in the dataset?

There are many open source data visualization tools available, some of which are listed here (many others are available):

- Matplotlib
- Seaborn
- Bokeh
- YellowBrick
- Tableau
- D3.js (JavaScript and SVG)

Incidentally, in case you have not already done so, it would be helpful to install the following Python libraries (using `pip3`) on your computer so that you can launch the code samples in this chapter:

```
pip3 install matplotlib
pip3 install seaborn
pip3 install bokeh
```

Types of Data Visualization

Bar graphs, line graphs, and pie charts are common ways to present data, and yet many other types exist, some of which are listed below:

- `2D/3D Area Chart`
- `Bar Chart`
- `Gantt Chart`
- `Heat Map`

- `Histogram`
- `Polar Area`
- `Scatter Plot (2D or 3D)`
- `Timeline`

The Python code samples in the next several sections illustrate how to perform visualization via rudimentary APIs from `matplotlib`.

WHAT IS MATPLOTLIB?

Matplotlib is a plotting library that supports NumPy, SciPy, and toolkits such as wxPython (among others). Matplotlib supports only version 3 of Python: support for version 2 of Python was available only through 2020. Matplotlib is a multi-platform library that is built on NumPy arrays.

The plotting-related code samples in this chapter use pyplot, which is a Matplotlib module that provides a MATLAB-like interface. Here is an example of using pyplot (copied from *https://www.biorxiv.org/content/10.1101/120378v1.full.pdf*) to plot a smooth curve based on negative powers of Euler's constant e:

```
import matplotlib.pyplot as plt
import numpy as np

a = np.linspace(0, 10, 100)
b = np.exp(-a)
plt.plot(a, b)
plt.show()
```

The code samples that plot line segments assume that you are familiar with the equation of a (non-vertical) line in the plane: `y = m*x + b`, where `m` is the slope and `b` is the y-intercept.

Furthermore, some code samples use NumPy APIs such as `np.linspace()`, `np.array()`, `np.random.rand()`, and `np.ones()` (discussed in Chapter 3), so you can refresh your memory regarding these APIs.

MATPLOTLIB STYLES

Listing 5.1 displays the content of `mpl_styles.py` that illustrates how to plot a pie chart in Matplotlib.

Listing 5.1: mpl_styles.py

```
import matplotlib.pyplot as plt

print("plt.style.available:")
styles = plt.style.available

for style in styles:
  print("style:",style)
```

Listing 5.1 contains an `import` statement, followed by the variable `styles` that is initialized with the set of available styles in Matplotlib. The final portion of Listing 5.1 contains a loop that iterates through the values in the `styles` variable. Launch the code in Listing 5.1, and you will see the following output:

```
plt.style.available:
style: Solarize_Light2
style: _classic_test_patch
style: bmh
style: classic
```

```
style: dark_background
style: fast
style: fivethirtyeight
style: ggplot
style: grayscale
style: seaborn
style: seaborn-bright
style: seaborn-colorblind
style: seaborn-dark
style: seaborn-dark-palette
style: seaborn-darkgrid
style: seaborn-deep
style: seaborn-muted
style: seaborn-notebook
style: seaborn-paper
style: seaborn-pastel
style: seaborn-poster
style: seaborn-talk
style: seaborn-ticks
style: sea born-white
style: seaborn-whitegrid
style: tableau-colorblind10
```

DISPLAY ATTRIBUTE VALUES

Listing 5.2 displays the content of `mat_attrib_values.py` that shows the attribute values of an object in Matplotlib (subplots are discussed later in this chapter).

Listing 5.2: mat_attrib_values.py

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

print("=> attribute values:")
print(plt.getp(fig))
```

Listing 5.2 contains an `import` statement, followed by the variables `fig` and `ax` that are initialized by invoking the `subplots()` method of the `plt` class. The next block of code prints the attribute values in `fig` by invoking the `plt.getp()` method. Launch the code in Listing 5.2, and you will see the following output:

```
=> attribute values:
    agg_filter = None
    alpha = None
    animated = False
    axes = [<AxesSubplot:>]
    children = [<matplotlib.patches.Rectangle
    object at 0x11c34f0...
    clip_box = None
    clip_on = True
    clip_path = None
    constrained_layout = False
    constrained_layout_pads = (0.04167, 0.04167,
    0.02, 0.02)
    contains = None
    default_bbox_extra_artists = [<AxesSubplot:>,
    <matplotlib.spines.Spine object a...
    dpi = 100.0
    edgecolor = (1.0, 1.0, 1.0, 1.0)
    facecolor = (1.0, 1.0, 1.0, 1.0)
    figheight = 4.8
    figure = None
    figwidth = 6.4
    frameon = True
    gid = None
    in_layout = True
    label =
    path_effects = []
    picker = None
    rasterized = None
    size_inches = [6.4 4.8]
    sketch_params = None
    snap = None
    tight_layout = False
    transform = IdentityTransform()
    transformed_clip_path_and_affine = (None, None)
    url = None
    visible = True
    window_extent = TransformedBbox( Bbox(x0=0.0,
    y0=0.0, x1=6.4, ...
    zorder = 0
None
```

COLOR VALUES IN MATPLOTLIB

Listing 5.3 displays the content of `mat_colors.py` that shows the colors available in Matplotlib.

Listing 5.3: mat_colors.py

```
import matplotlib
import matplotlib.pyplot as plt

colors = plt.colormaps()

col_count=5
idx=0
for color in colors:
  if(color.endswith("_r") == False):
    print(color," ",end="")
    idx += 1
    if(idx % col_count == 0):
      print()
print()
print("=> color count:",idx)
```

Listing 5.3 contains two `import` statements, after which the variable `colors` is initialized with the list of available colors. The next portion of Listing 5.3 contains a loop that iterates through the `colors` variable and prints the value of each color, provided that it does not have the suffix "`_r`" in its name. A new line is printed each time that five colors have been printed. Launch the code in Listing 5.3, and you will see the following output:

```
Accent Blues BrBG BuGn BuPu
CMRmap Dark2 GnBu Greens Greys
OrRd Oranges PRGn Paired Pastel1
Pastel2 PiYG PuBu PuBuGn PuOr
PuRd Purples RdBu RdGy RdPu
RdYlBu RdYlGn Reds Set1 Set2
Set3 Spectral Wistia YlGn YlGnBu
```

```
YlOrBr YlOrRd afmhot autumn binary
bone brg bwr cividis cool
coolwarm copper cubehelix flag gist_earth
gist_gray gist_heat gist_ncar gist_rainbow
gist_stern
gist_yarg gnuplot gnuplot2 gray hot
hsv inferno jet magma nipy_spectral
ocean pink plasma prism rainbow
seismic spring summer tab10 tab20
tab20b tab20c terrain turbo twilight
twilight_shifted viridis winter
=> color count: 83
```

Let’s proceed to the next section that contains a set of basic code samples that display various types of line segments.

CUBED NUMBERS IN MATPLOTLIB

Listing 5.4 displays the content of `cubed_numbers.py` that illustrates how to plot a set of points using Matplotlib.

Listing 5.4: cubed_numbers.py

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4], [1, 8, 27, 64])
plt.axis([0, 5, 0, 70])
plt.xlabel("Integers (1-4)")
plt.ylabel("Cubed Integers")
plt.show()
```

Listing 5.4 plots a set of integer-valued points whose x-coordinate is between 1 and 4 inclusive and whose y-coordinate is the cube of the corresponding x-coordinate. The code sample also labels the horizontal axis and the vertical axis. Figure 5.1 displays these points in Listing 5.4.

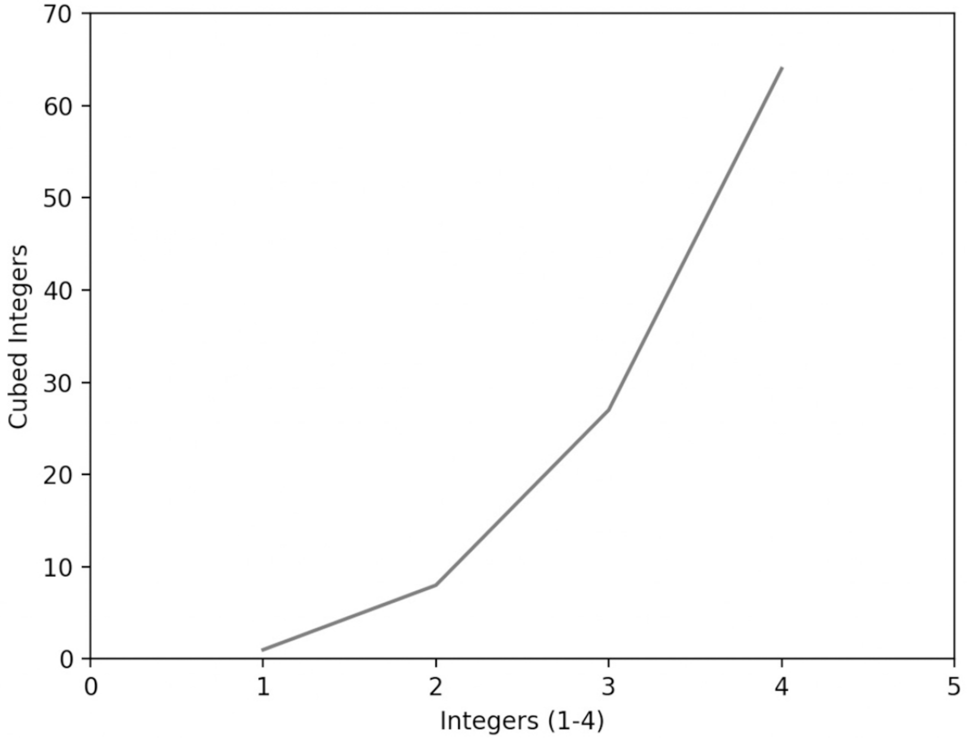


FIGURE 5.1 A graph of cubed numbers

HORIZONTAL LINES IN MATPLOTLIB

Listing 5.5 displays the content of `hlines1.py` that illustrates how to plot horizontal lines using Matplotlib. Recall that the equation of a non-vertical line in the 2D plane is `y = m*x + b`, where `m` is the slope of the line and `b` is the y-intercept of the line.

Listing 5.5: hlines1.py

```
import numpy as np
import matplotlib.pyplot as plt

# top line
x1 = np.linspace(-5,5,num=200)
y1 = 4 + 0*x1

# middle line
x2 = np.linspace(-5,5,num=200)
y2 = 0 + 0*x2

# bottom line
x3 = np.linspace(-5,5,num=200)
y3 = -3 + 0*x3

plt.axis([-5, 5, -5, 5])
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.plot(x3,y3)
plt.show()
```

Listing 5.5 uses the `np.linspace()` API to generate a list of 200 equally spaced numbers for the horizontal axis, all of which are between -5 and 5. The three lines defined via the variables `y1`, `y2`, and `y3`, are defined in terms of the variables `x1`, `x2`, and `x3`, respectively.

Figure 5.2 displays three horizontal line segments whose equations are contained in Listing 5.5.

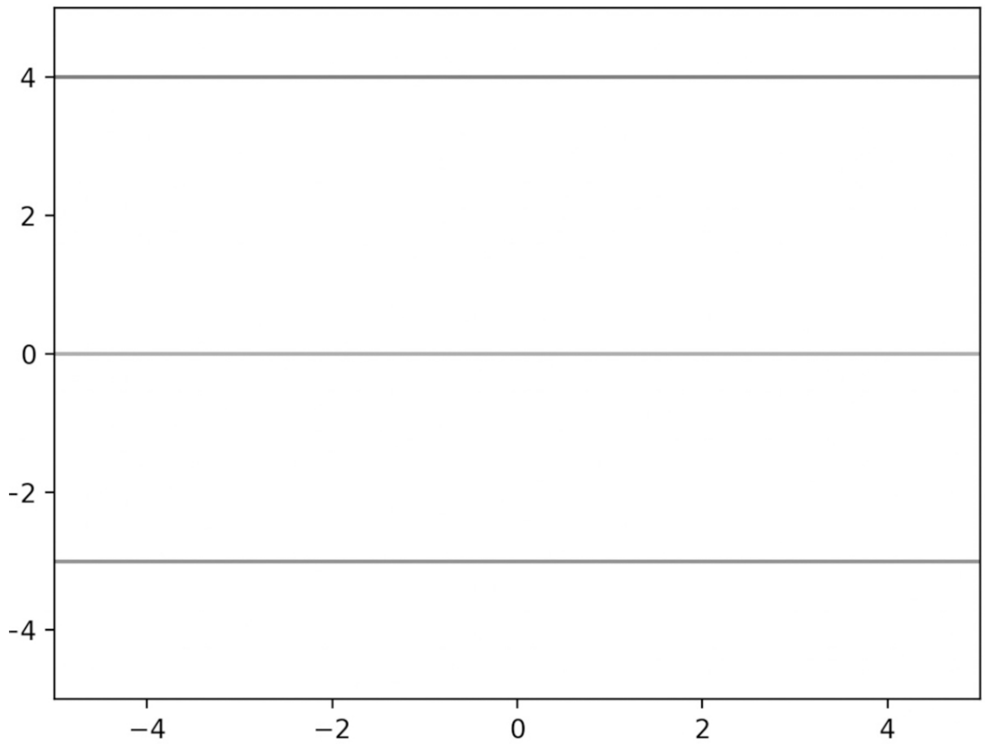


FIGURE 5.2 A graph of three horizontal line segments

SLANTED LINES IN MATPLOTLIB

Listing 5.6 displays the content of `diagonallines.py` that illustrates how to plot slanted lines.

Listing 5.6: diagonallines.py

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.linspace(-5,5,num=200)
y1 = x1

x2 = np.linspace(-5,5,num=200)
y2 = -x2
```

```
plt.axis([-5, 5, -5, 5])
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.show()
```

Listing 5.6 defines two lines using the technique that you saw in Listing 5.5, except that these two lines define `y1 = x1` and `y2 = -x2`, which produces slanted lines instead of horizontal lines.

Figure 5.3 shows two slanted line segments whose equations are defined in Listing 5.6.

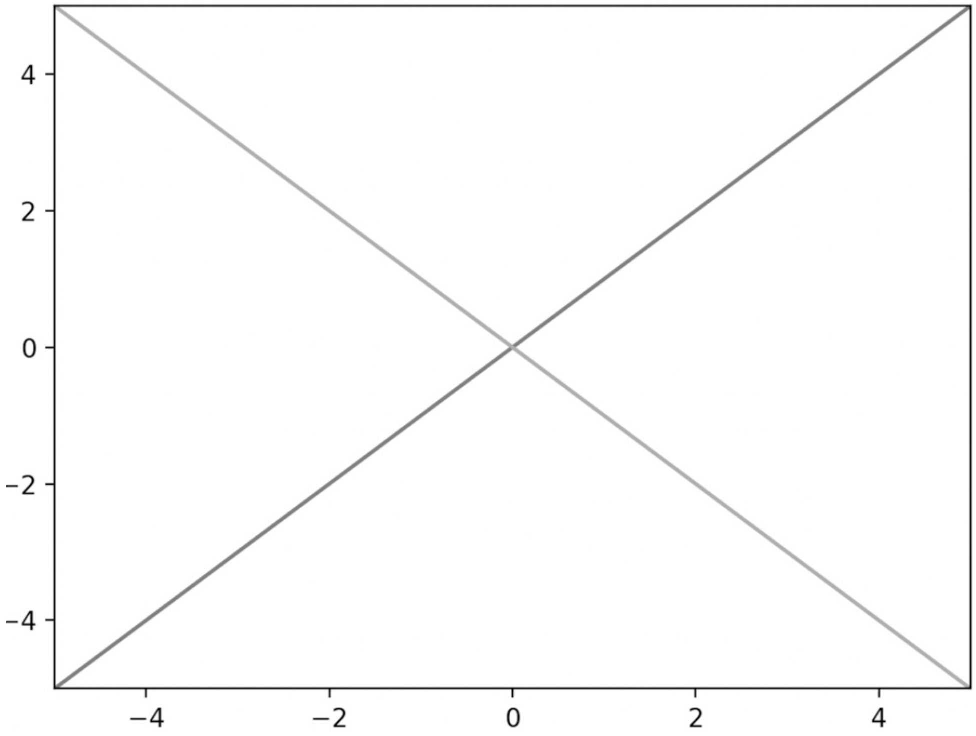


FIGURE 5.3 A graph of two slanted line segments

PARALLEL SLANTED LINES IN MATPLOTLIB

If two lines in the Euclidean plane have the same slope, then they are parallel. Listing 5.7 displays the content of `parallellines1.py` that illustrates how to plot parallel slanted lines.

Listing 5.7: parallellines1.py

```
import matplotlib.pyplot as plt
import numpy as np

# lower line
x1 = np.linspace(-5,5,num=200)
y1 = 2*x1

# upper line
x2 = np.linspace(-5,5,num=200)
y2 = 2*x2 + 3

# horizontal axis
x3 = np.linspace(-5,5,num=200)
y3 = 0*x3 + 0

# vertical axis
plt.axvline(x=0.0)

plt.axis([-5, 5, -10, 10])
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.plot(x3,y3)
plt.show()
```

Listing 5.7 defines three lines using the technique that you saw in Listing 5.6, where these three lines are slanted and also parallel to each other.

Figure 5.4 displays two slanted and also parallel line segments whose equations are defined in Listing 5.4.

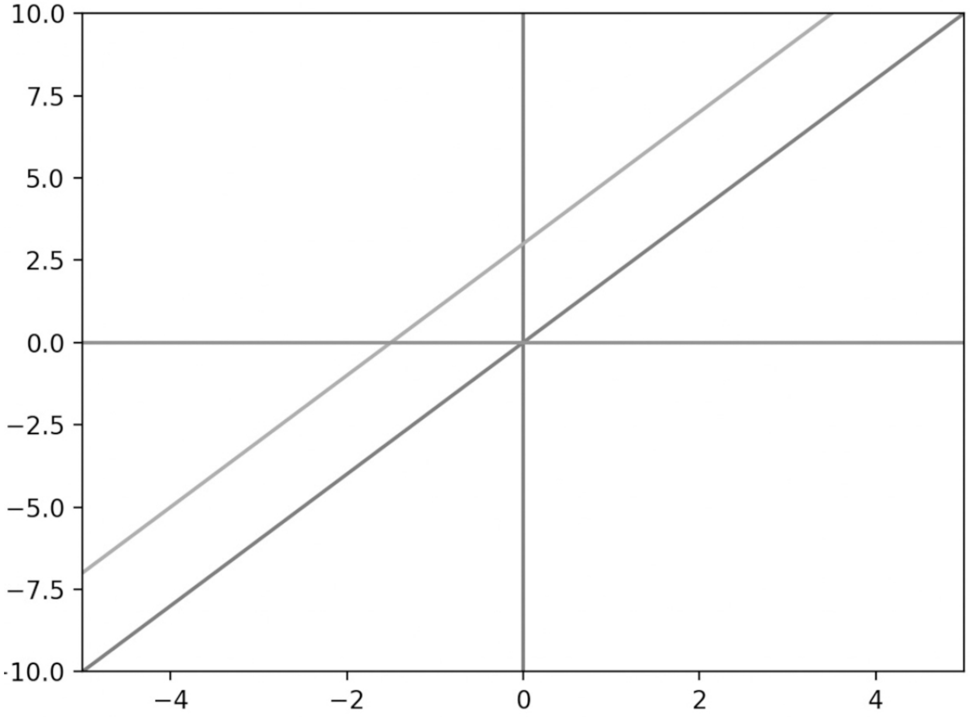


FIGURE 5.4 A graph of two slanted parallel line segments

LINES AND LABELED VERTICES IN MATPLOTLIB

Listing 5.8 displays the content of `multi_lines.py` that illustrates how to plot multiple line segments with labeled vertices.

Listing 5.8: multi_lines.py

```
import matplotlib.pyplot as plt

x_coord = [ 50, 300, 175, 50]
y_coord = [ 50, 50, 150, 50]
plt.plot(x_coord,y_coord)
plt.scatter(x_coord,y_coord)

for x,y in zip(x_coord,y_coord):
  plt.text(x,y,'Coord ({x},{y})'.format(x=x,y=y))

x_coord = [ 175, 300, 50, 175]
y_coord = [ 50, 150, 150, 50]
plt.plot(x_coord,y_coord)
plt.scatter(x_coord,y_coord)

for x,y in zip(x_coord,y_coord):
  plt.text(x,y,'Coord ({x},{y})'.format(x=x,y=y))
plt.show()
```

Listing 5.8 defines the NumPy variable `points` that defines a 2D list of points with three rows and four columns. The Pyplot API `plot()` uses the `points` variable to display a grid-like pattern. Figure 5.5 shows the grid of points defined in Listing 5.9.

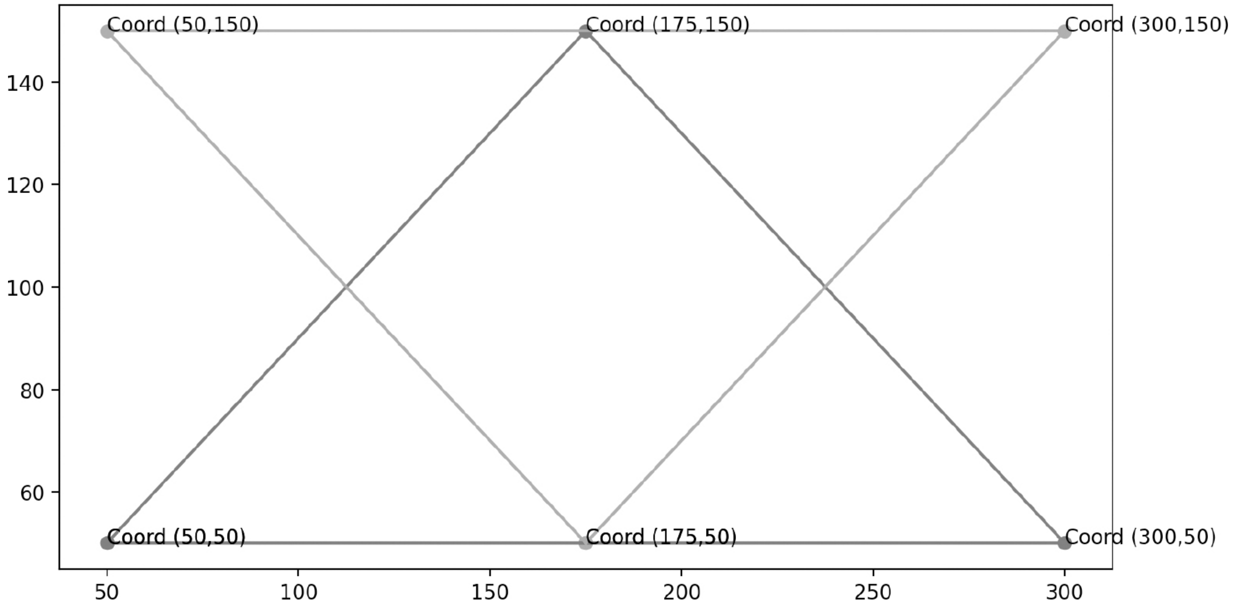


FIGURE 5.5 Lines and labeled vertices

A DOTTED GRID IN MATPLOTLIB

Listing 5.9 displays the content of `plotdottedgrid1.py` that illustrates how to plot a “dotted” grid pattern.

Listing 5.9: plotdottedgrid1.py

```
import numpy as np
import pylab
from itertools import product
import matplotlib.pyplot as plt

fig = pylab.figure()
ax = fig.add_subplot(1,1,1)

ax.grid(which='major', axis='both', linestyle='--
')

[line.set_zorder(3) for line in ax.lines]
fig.show() # to update

plt.gca().xaxis.grid(True)
plt.show()
```

Listing 5.9 is similar to the code in Listing 5.8 in that both of them plot a grid-like pattern; however, the former renders a “dotted” grid pattern whereas the latter renders a “dotted” grid pattern by specifying the value `'--'` for the `linestyle` parameter.

The next portion of Listing 5.9 invokes the `set_zorder()` method that controls which items are displayed on top of other items, such as dots on top of lines, or vice versa. The final portion of Listing 5.9 invokes the `gca().xaxis.grid(True)` chained methods to display the vertical grid lines.

You can also use the `plt.style` directive to specify a style for figures. The following code snippet specifies the classic style of Matplotlib:

```
plt.style.use('classic')
```

Figure 5.6 shows a “dashed” grid pattern based on the code in Listing 5.10.

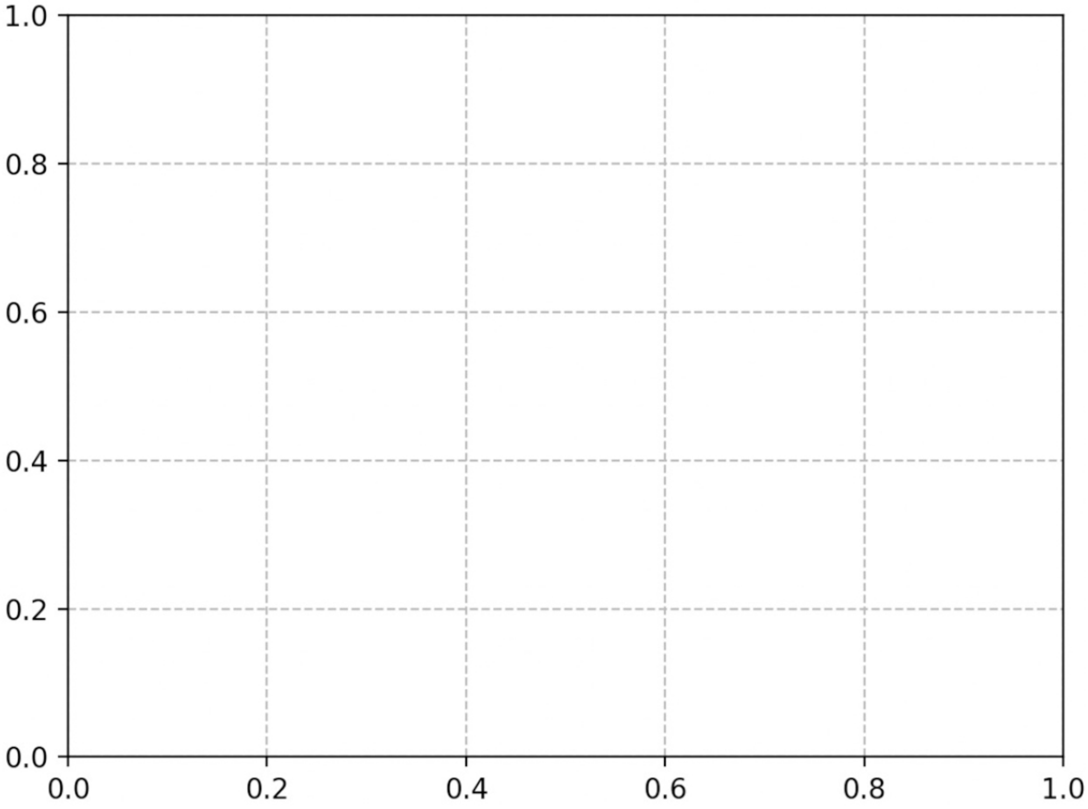


FIGURE 5.6 A “dashed” grid pattern

LINES IN A GRID IN MATPLOTLIB

Listing 5.10 displays the content of `plotlinegrid2.py` that illustrates how to plot lines in a grid.

Listing 5.10: plotlinegrid2.py

```
import numpy as np
import pylab
from itertools import product
import matplotlib.pyplot as plt

fig = plt.figure()
graph = fig.add_subplot(1,1,1)
graph.grid(which='major', linestyle='-',
linewidth='0.5', color='red')

x1 = np.linspace(-5,5,num=200)
y1 = 1*x1
graph.plot(x1,y1, 'r-o')

x2 = np.linspace(-5,5,num=200)
y2 = -x2
graph.plot(x2,y2, 'b-x')

fig.show() # to update
plt.show()
```

Listing 5.10 defines the NumPy variable `points` that defines a 2D list of points with three rows and four columns. The Pyplot API `plot()` uses the `points` variable to display a grid-like pattern.

Figure 5.7 displays a set of “dashed” line segment whose equations are contained in Listing 5.10.

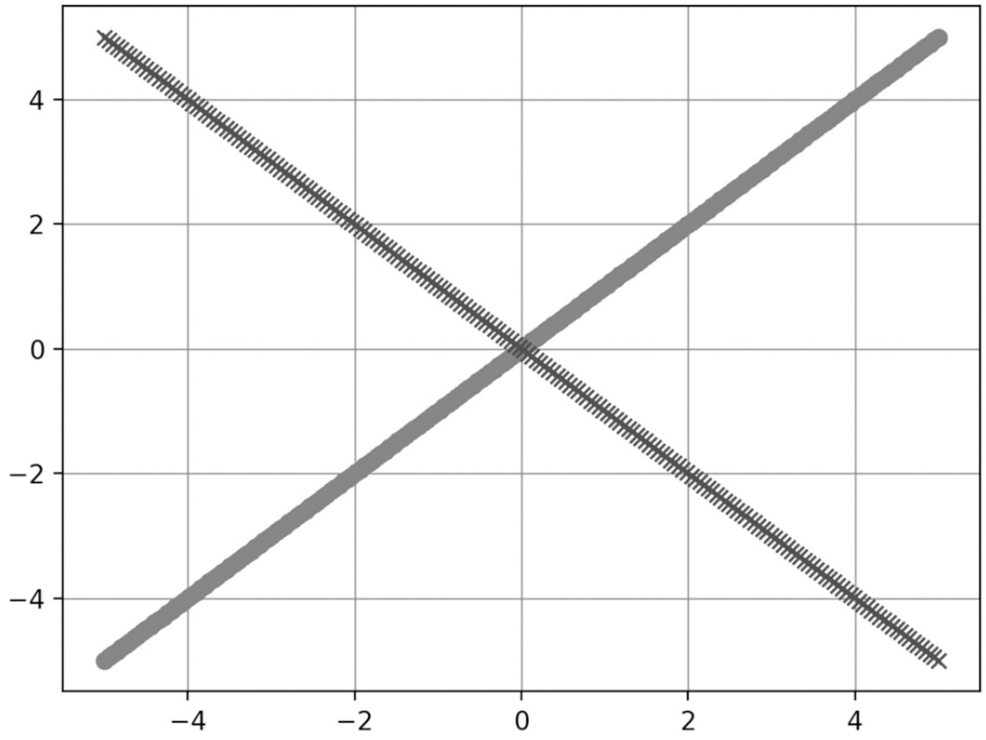


FIGURE 5.7 A grid of line segments

TWO LINES AND A LEGEND IN MATPLOTLIB

Listing 5.11 displays the content of `plotgrid2.py` that illustrates how to display a colored grid.

Listing 5.11: two_lines_legend.py

```
import matplotlib.pyplot as plt

# FIRST PLOT:
vals_x = [91,93,95,96,97,98,99,99,104,115]
vals_y =
[1500,2000,3000,2500,1200,1500,2900,3200,5200,6500
]
```

```
plt.plot(vals_x, vals_y) # alternate style
#plt.plot(vals_x, vals_y, label='First List')

# SECOND PLOT:
vals_x2 = [91,93,95,96,97,98,99,99,104,115]
vals_y2 =
[1005,1006,1007,1008,1009,2031,3100,2033,3034,4035
]
plt.plot(vals_x2, vals_y2)
#plt.plot(vals_x2, vals_y2, label='Second List') #
alternate style

# generate line plot:
plt.plot(vals_x, vals_y)
plt.title("Random Pairs of Numbers")
plt.xlabel("Random X Values")
plt.ylabel("Random Y Values")
plt.legend(['First List','Second List'])
#plt.legend() # alternate style
plt.show()
```

Listing 5.11 defines the NumPy variable `data`, which defines a 2D set of points with ten rows and ten columns. The Pyplot API `plot()` uses the `data` variable to display a colored grid-like pattern. Figure 5.8 displays a colored grid whose equations are contained in Listing 5.11.

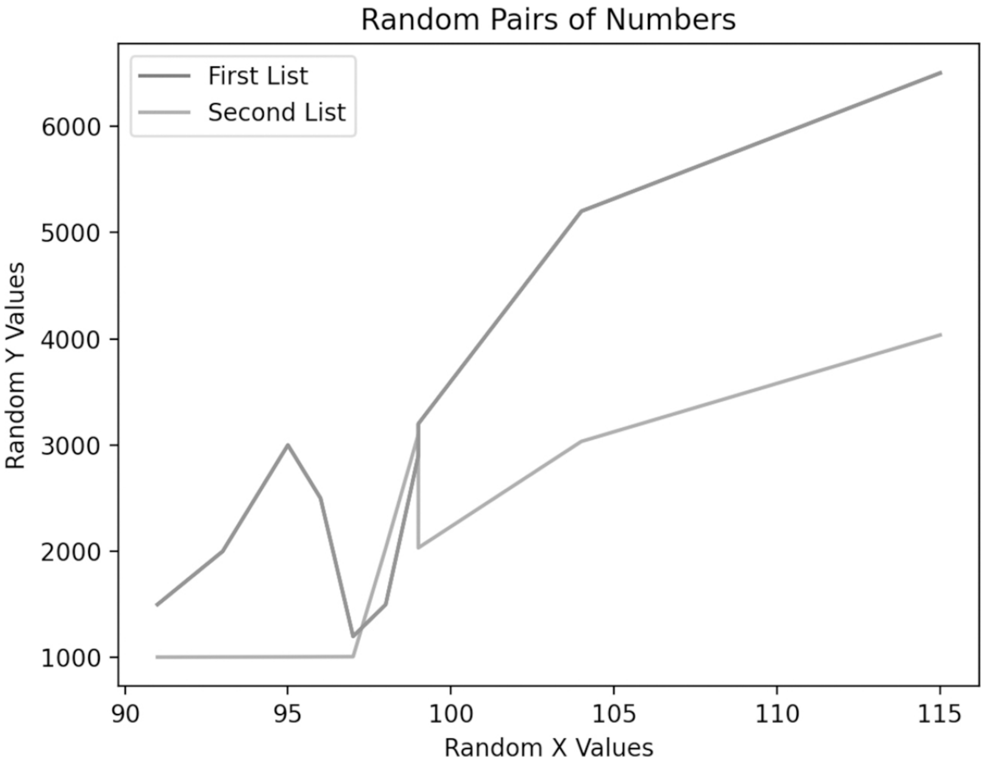


FIGURE 5.8 Two lines and a legend

LOADING IMAGES IN MATPLOTLIB

Listing 5.12 displays the content of `load_images2.py` that illustrates how to display an image.

Listing 5.12: load_images2.py

```
from sklearn.datasets import load_digits
from matplotlib import pyplot as plt

digits = load_digits()
#set interpolation='none'
fig = plt.figure(figsize=(3, 3))
plt.imshow(digits['images'][66], cmap="gray",
interpolation='none')
```

```
plt.show()
```

Listing 5.12 starts with two `import` statements and then the `digits` variable is initialized with the contents of the `digits` dataset. The next portion of Listing 5.12 displays the content of one of the images in the `digits` dataset. Launch the code in Listing 5.12, and you will see the image in Figure 5.9.

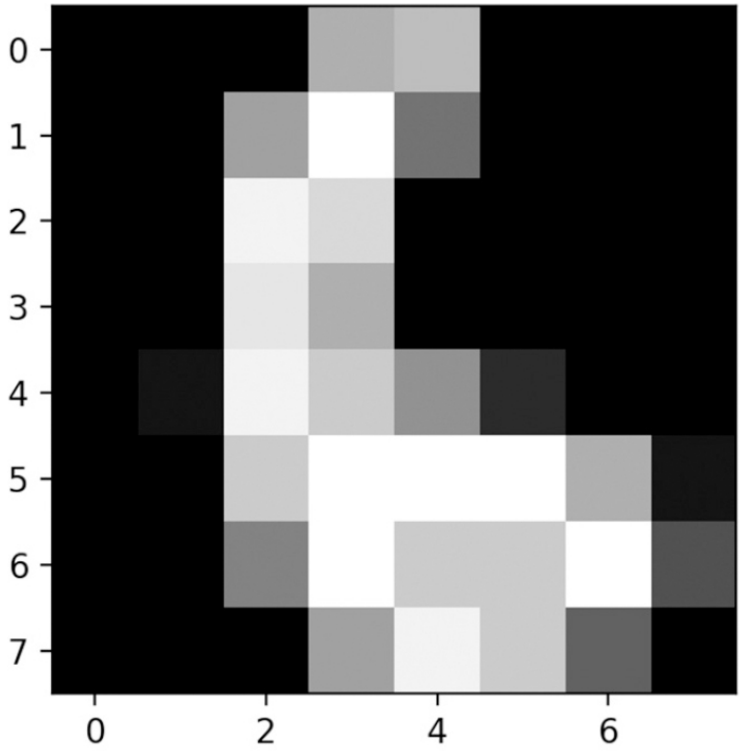


FIGURE 5.9 Loading an image in Matplotlib

A CHECKERBOARD IN MATPLOTLIB

Listing 5.13 displays the content of `checkerboard1.py` that illustrates how to display a checkerboard.

Listing 5.13: checkerboard1.py

```
import matplotlib.pyplot as plt
from matplotlib import colors
import numpy as np

data = np.random.rand(10, 10) * 20

# create discrete colormap
cmap = colors.ListedColormap(['red', 'blue'])
bounds = [0,10,20]
norm = colors.BoundaryNorm(bounds, cmap.N)

fig, ax = plt.subplots()
ax.imshow(data, cmap=cmap, norm=norm)

# draw gridlines
ax.grid(which='major', axis='both', linestyle='-',
color='k', linewidth=2)
ax.set_xticks(np.arange(-.5, 10, 1));
ax.set_yticks(np.arange(-.5, 10, 1));

plt.show()
```

Listing 5.13 defines the NumPy variable `data` that defines a 2D set of points with ten rows and ten columns. The Pyplot API `plot()` uses the `data` variable to display a colored grid-like pattern. Figure 5.10 shows a colored grid whose equations are contained in Listing 5.13.

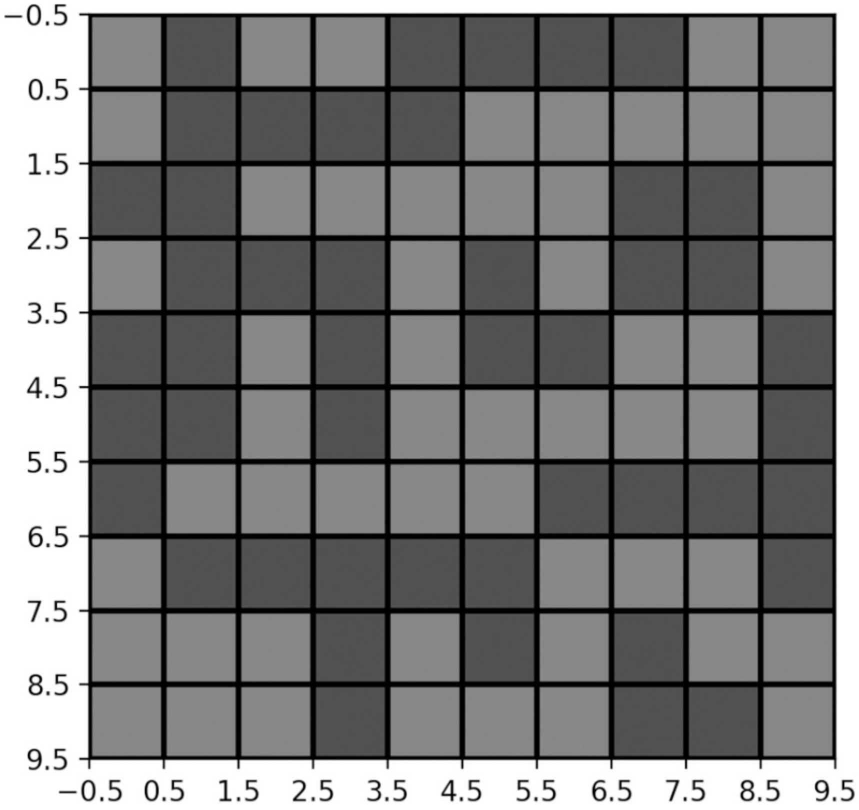


FIGURE 5.10 A checkerboard

RANDOMIZED DATA POINTS IN MATPLOTLIB

Listing 5.14 displays the content of `lin_reg_plot.py` that illustrates how to plot a graph of random points.

Listing 5.14: lin_plot_reg.py

```
import numpy as np
import matplotlib.pyplot as plt

trX = np.linspace(-1, 1, 101) # Linear space of
101 and [-1,1]
```

```
#Create the y function based on the x axis
trY = 2*trX + np.random.randn(*trX.shape)*0.4+0.2

#create figure and scatter plot of the random
points
plt.figure()
plt.scatter(trX,trY)

# Draw one line with the line function
plt.plot (trX, .2 + 2 * trX)
plt.show()
```

Listing 5.14 defines the NumPy variable `trX` that contains 101 equally spaced numbers that are between -1 and 1 (inclusive). The variable `trY` is defined in two parts: the first part is `2*trX` and the second part is a random value that is partially based on the length of the one-dimensional array `trX`. The variable `trY` is the sum of these two “parts,” which creates a “fuzzy” line segment.

The next portion of Listing 5.14 creates a scatterplot based on the values in `trX` and `trY`, followed by the Pyplot API `plot()` that renders a line segment. Figure 5.11 shows a random set of points based on the code in Listing 5.14.

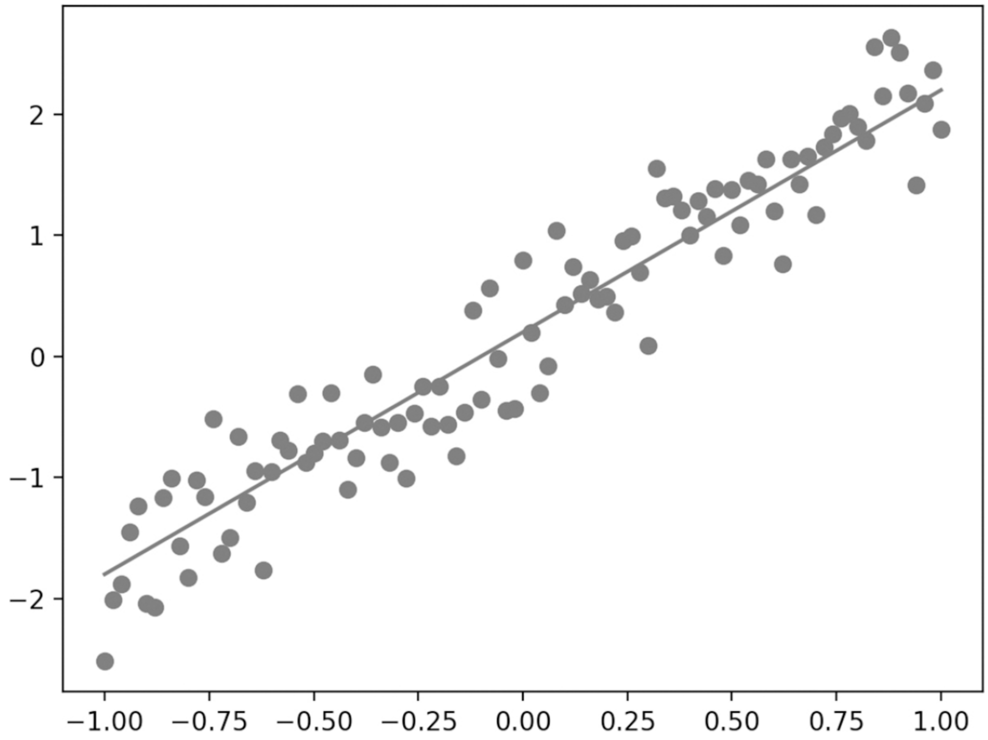


FIGURE 5.11 A random set of points

A SET OF LINE SEGMENTS IN MATPLOTLIB

Listing 5.15 displays the content of `line_segments.py` that illustrates how to plot a set of connected line segments in Matplotlib.

Listing 5.15: line_segments.py

```
import numpy as np
import matplotlib.pyplot as plt

x = [7,11,13,15,17,19,23,29,31,37]

plt.plot(x) # OR: plt.plot(x, 'ro-') or bo
plt.ylabel('Height')
plt.xlabel('Weight')
plt.show()
```

Listing 5.15 defines the array `x` that contains a hard-coded set of values. The Pyplot API `plot()` uses the variable `x` to display a set of connected line segments. Figure 5.12 shows the result of launching the code in Listing 5.16.

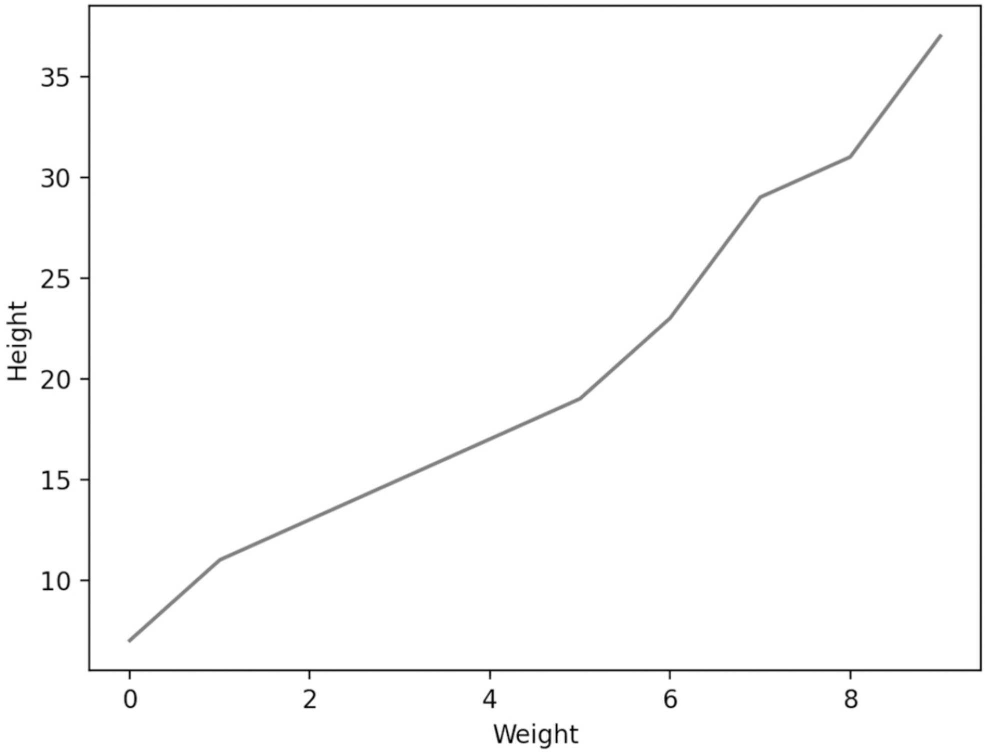


FIGURE 5.12 A set of connected line segments

PLOTTING MULTIPLE LINES IN MATPLOTLIB

Listing 5.16 displays the content of `plt_array2.py` that illustrates the ease with which you can plot multiple lines in Matplotlib.

Listing 5.16: plt_array2.py

```
import matplotlib.pyplot as plt
```

```
x = [7,11,13,15,17,19,23,29,31,37]
data = [[8, 4, 1], [5, 3, 3], [6, 0, 2], [1, 7,
9]]
plt.plot(data, 'd-')
plt.show()
```

Listing 5.16 defines the array `data` that contains a hard-coded set of values. The Pyplot API `plot()` uses the variable `data` to display a line segment. Figure 5.13 shows multiple lines based on the code in Listing 5.16.

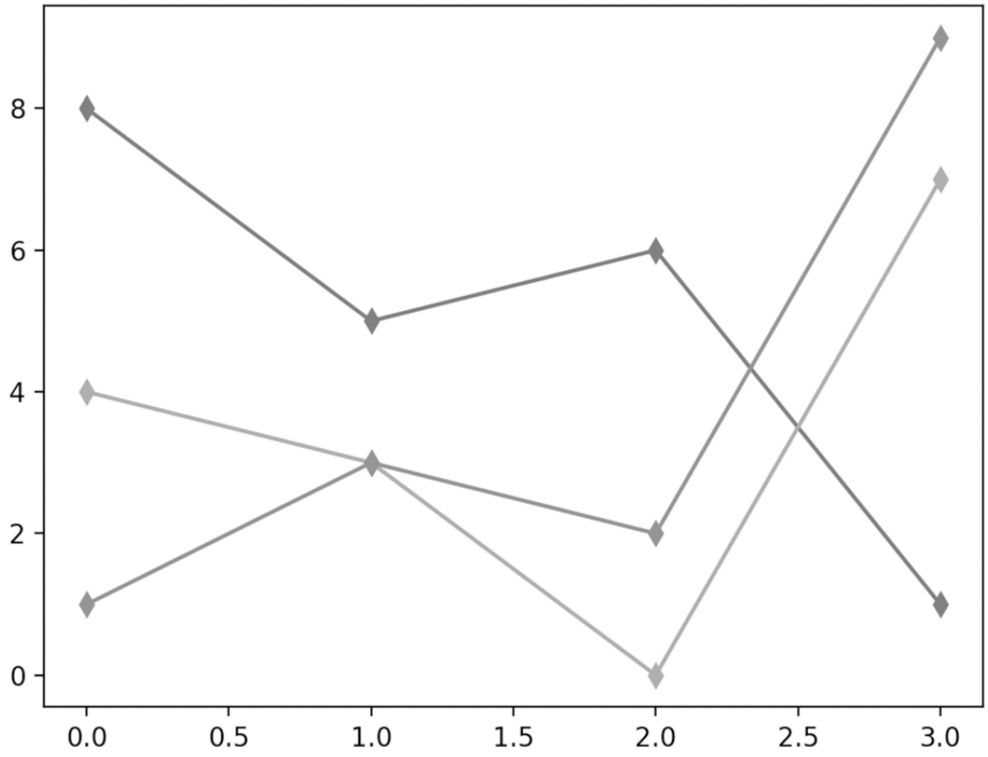


FIGURE 5.13 Multiple lines in Matplotlib

TRIGONOMETRIC FUNCTIONS IN MATPLOTLIB

You can display the graph of trigonometric functions as easily as you can render “regular” graphs using Matplotlib. Listing 5.17 displays the content of `sincos.py` that illustrates how to plot a sine function and a cosine function in Matplotlib.

Listing 5.17: sincos.py

```
import numpy as np
import math

x = np.linspace(0, 2* math.pi, 101)
s = np.sin(x)
c = np.cos(x)

import matplotlib.pyplot as plt
plt.plot (s)
plt.plot (c)
plt.show()
```

Listing 5.17 defines the NumPy variables `x`, `s`, and `c` using the NumPy APIs `linspace()`, `sin()`, and `cos()`, respectively. Next, the Pyplot API `plot()` uses these variables to display a sine function and a cosine function.

Figure 5.14 shows a graph of two trigonometric functions based on the code in Listing 5.17.

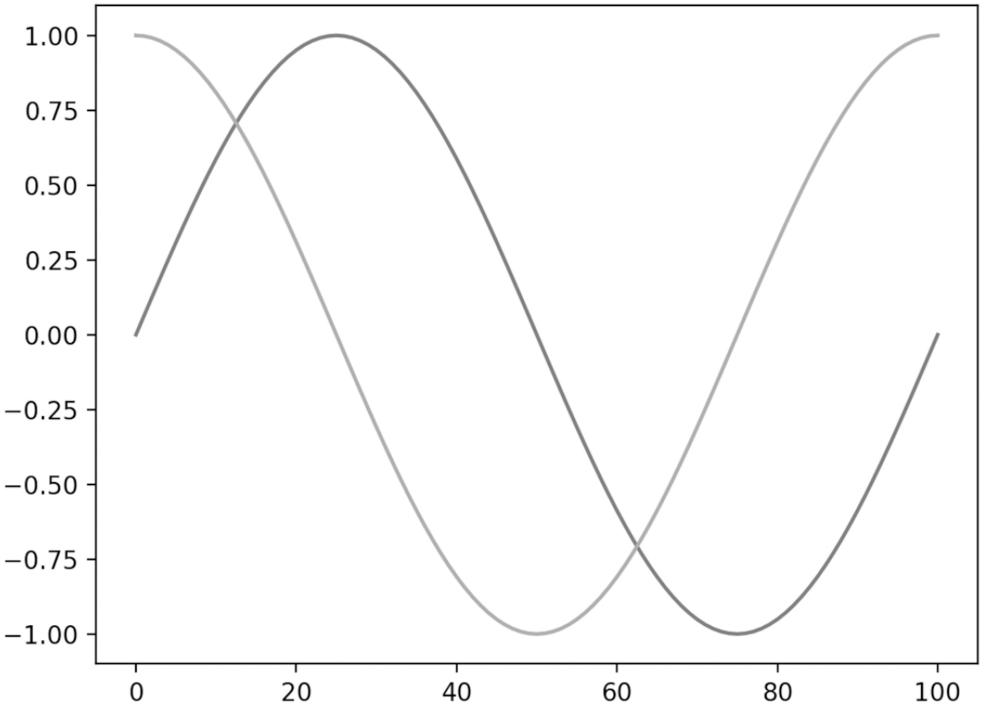


FIGURE 5.14 Sine and cosine functions

Let’s look at a simple dataset consisting of discrete data points, which is the topic of the next section.

A HISTOGRAM IN MATPLOTLIB

Listing 5.18 displays the content of `histogram1.py` that illustrates how to plot a histogram using Matplotlib.

Listing 5.18: histogram1.py

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5, 6, 7, 4]

plt.hist(x, bins = [1, 2, 3, 4, 5, 6, 7])
plt.title("Histogram")
plt.legend(["bar"])
plt.show()
```

Listing 5.18 is straightforward: the variable `x` is initialized as a set of numbers, followed by a block of code that renders a histogram based on the data in the variable `x`. Launch the code in Listing 5.18, and you will see the histogram that is shown in Figure 5.15.

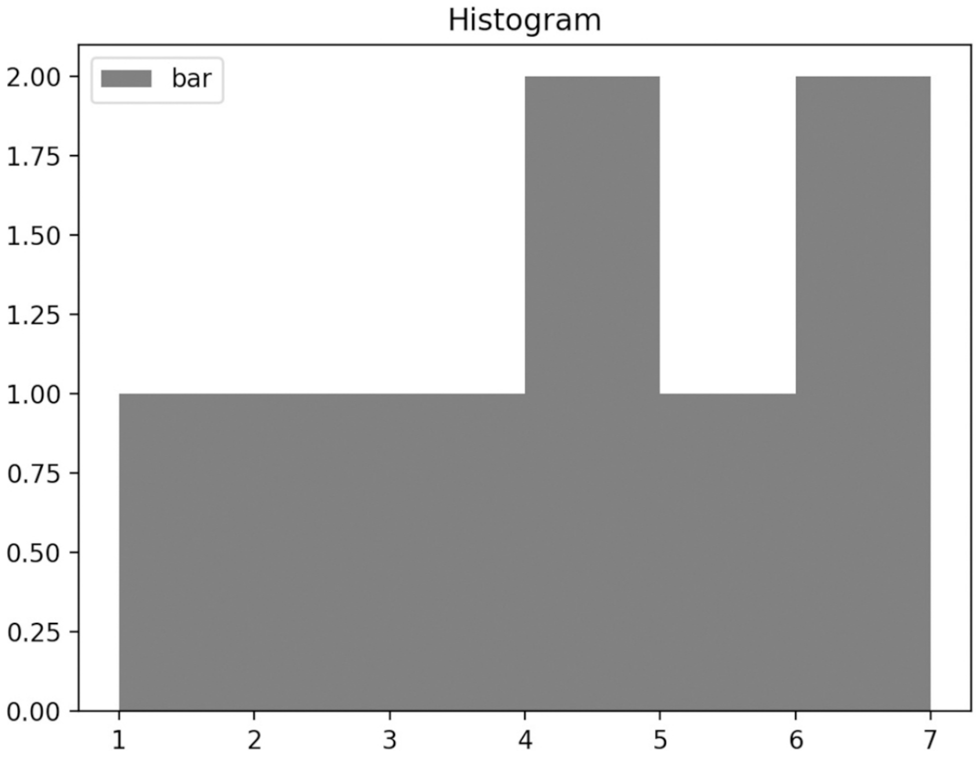


FIGURE 5.15 A histogram based on random values

HISTOGRAM WITH DATA FROM A SQLITE3 TABLE

Listing 5.19 displays the content of `rainfall_hist3.py` that shows you how to define a simple SQL query to create a histogram based on the data from the rainfall table.

Listing 5.19: rainfall_hist3.py

```
import sqlite3
import pandas as pd
import matplotlib.pyplot as plt

sql = """
  SELECT
  cast(centimeters/5.00 as int)*5 as cent_floor,
  count(*) as count
FROM rainfall
GROUP by 1
ORDER by 1;
"""

con = sqlite3.connect ("rain.db")
df = pd.read_sql_query(sql, con)
con.close()

print("=> Histogram of Rainfall:")
print(df)

#df.hist(column='count', bins=7, grid=False,
rwidth=1.0, color='red')
df.hist(column='count', bins=14, grid=False,
rwidth=.8, color='red')
plt.show()
```

Listing 5.19 starts with several `import` statements and then initializes the variable `sql` with a SQL statement that selects data from the `rainfall` table. The next portion of Listing 5.19 initializes the variable `con` to access the appropriate data from the `rain.db` database, which is included in the supplemental files for this chapter. The next code snippet invokes the `read_sql_query()` method of Pandas to populate the Pandas data frame `df` with the data returned by executing the SQL statement contained in the variable `sql`. Launch the code in Listing 5.19, and you will see the following output:

```
=> Histogram of Rainfall:
   bucket_floor     bucket_name  count
0             0     FROM 0 TO 10     27
1            10   FROM 10 TO 20     14
2            20   FROM 20 TO 30      9
3            30   FROM 30 TO 40      9
4            40   FROM 40 TO 50      3
5            50   FROM 50 TO 60      2
6            60   FROM 60 TO 70      1
```

In addition to the preceding output, you will also see the histogram that is shown in Figure 5.16.

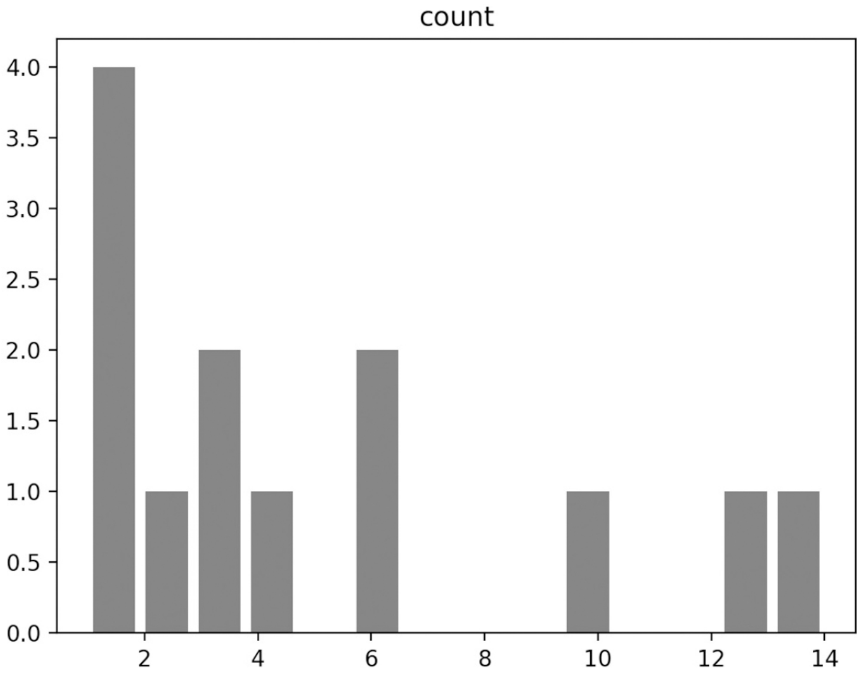


FIGURE 5.16 A histogram created from data from sqlite3

PLOT A BEST-FITTING LINE WITH GGPLOT

Listing 5.20 displays the content of `plot_best_fit.py` that illustrates how to plot a best-fitting line in Matplotlib.

Listing 5.20: plot_best_fit.py

```
import numpy as np

xs = np.array([1,2,3,4,5], dtype=np.float64)
ys = np.array([1,2,3,4,5], dtype=np.float64)

def best_fit_slope(xs,ys):
  m = (((np.mean(xs)*np.mean(ys))-np.mean(xs*ys)) /
       ((np.mean(xs)**2) - np.mean(xs**2)))
  b = np.mean(ys) - m * np.mean(xs)

  return m, b

m,b = best_fit_slope(xs,ys)
print('m:',m,'b:',b)

regression_line = [(m*x)+b for x in xs]

import matplotlib.pyplot as plt
from matplotlib import style
style.use('ggplot')

plt.scatter(xs,ys,color='#0000FF')
plt.plot(xs, regression_line)
plt.show()
```

Listing 5.20 defines the NumPy array variables `xs` and `ys` that are “fed” into the Python function `best_fit_slope()` that calculates the slope `m` and the y-intercept `b` for the best-fitting line. The Pyplot API `scatter()` displays a scatter plot of the points `xs` and `ys`, followed by the `plot()` API that displays the best-fitting line. Figure 5.17 shows a simple line based on the code in Listing 5.20.

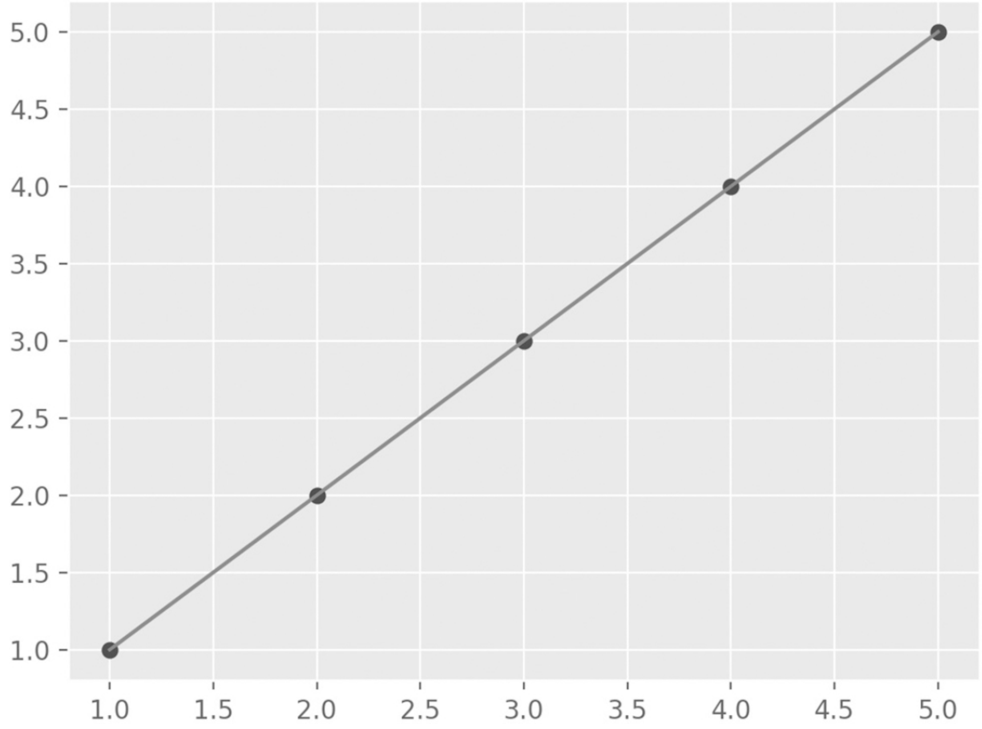


FIGURE 5.17 A best-fitting line for a 2D dataset

PLOT BAR CHARTS

Listing 5.21 displays the content of `barchart1.py` that illustrates how to plot a bar chart in Matplotlib.

Listing 5.21: barchart1.py

```
import matplotlib.pyplot as plt

x = [3, 1, 3, 12, 2, 4, 4]
```

```
y = [3, 2, 1, 4, 5, 6, 7]

plt.bar(x, y)
plt.title("Bar Chart")
plt.legend(["bar"])
plt.show()
```

Listing 5.21 contains an `import` statement followed by the variables `x` and `y` that are initialized as a list of numbers. Next, the bar chart is generated by invoking the `bar()` method of the `plt` class. The final block of code sets the title and legend for the bar chart and then displays the bar chart. Launch the code in Listing 5.21, and you will see the pie chart shown in Figure 5.18.

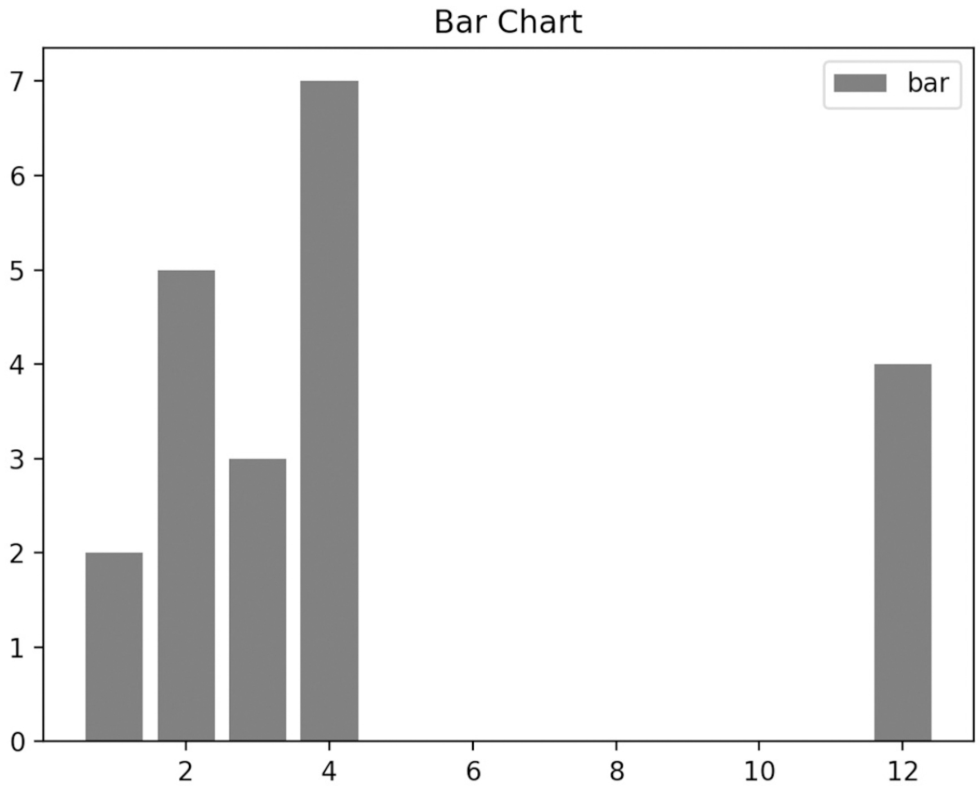


FIGURE 5.18 A bar chart from array data

Listing 5.22 displays the content of `barchart2.py` that illustrates how to plot a bar chart in Matplotlib.

Listing 5.22: barchart2.py

```
import matplotlib.pyplot as plt

plt.bar([0.25,1.25,2.25,3.25,4.25],
        [50,40,70,80,20],
        label="GDP1",width=.5)
plt.bar([.75,1.75,2.75,3.75,4.75],
        [80,20,20,50,60],
        label="GDP2", color='r',width=.5)

plt.legend()
plt.xlabel('Months')
plt.ylabel('GDP (Billion Euross)')
plt.title('Bar Chart Comparison')
```

Listing 5.22 contains an `import` statement followed by the definition of two bar charts that are displayed in a side-by-side manner. Notice that the definition of each bar chart involves specifying the `x` and `y` (even though they are not explicitly included), followed by a value for the `label` and `width` arguments. The final block of code sets the legend and labels for the horizontal and vertical axes. Launch the code in Listing 5.22, and you will see the pie chart shown in Figure 5.19.

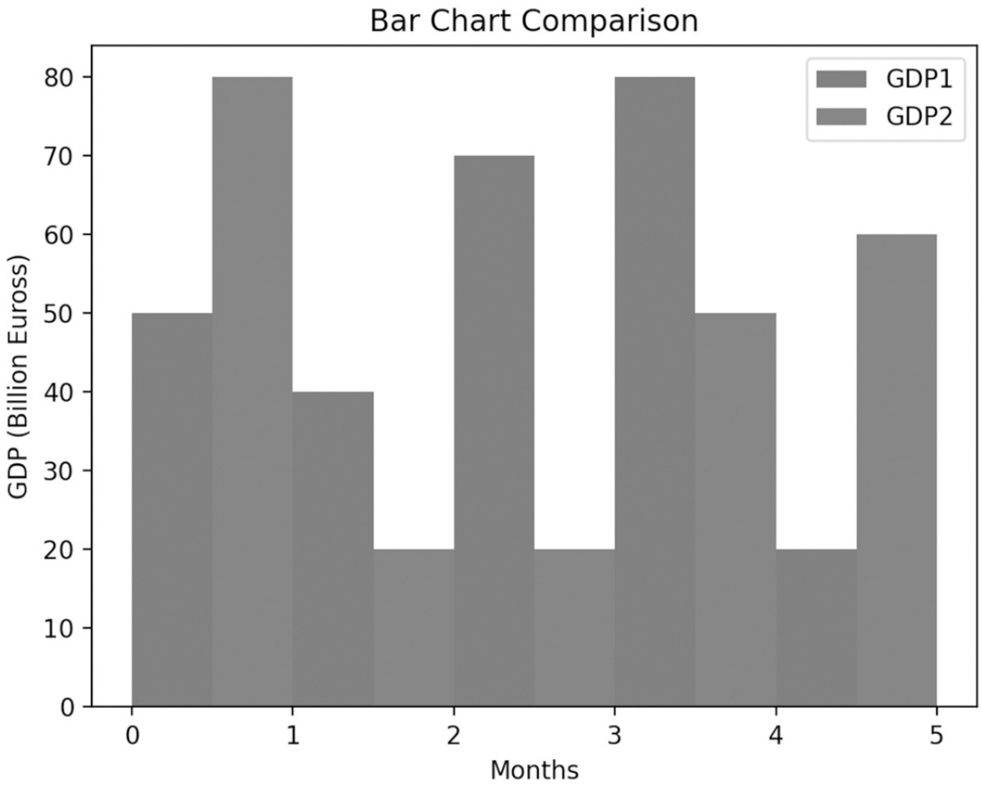


FIGURE 5.19 Two bar charts

PLOT A PIE CHART

Listing 5.23 displays the content of `piechart1.py` that illustrates how to plot a pie chart in Matplotlib.

Listing 5.23: piechart1.py

```
import numpy as np

# data to display on plots
x = [1, 2, 3, 4]

# explode the first wedge:
e =(0.1, 0, 0, 0)
```

```
plt.pie(x, explode = e)
plt.title("Pie chart")
plt.show()
```

Listing 5.23 contains an `import` statement followed by the variables `x` and `e` that are initialized as a list of numbers. The values for `x` are used to calculate the relative size of each “slice” of the pie chart, and the values for the variable `e` indicate that the first pie slice is “exploded” slightly (indicated by the value 0.1 in `e`), Launch the code in Listing 5.23, and you will see the pie chart displayed in Figure 5.20.

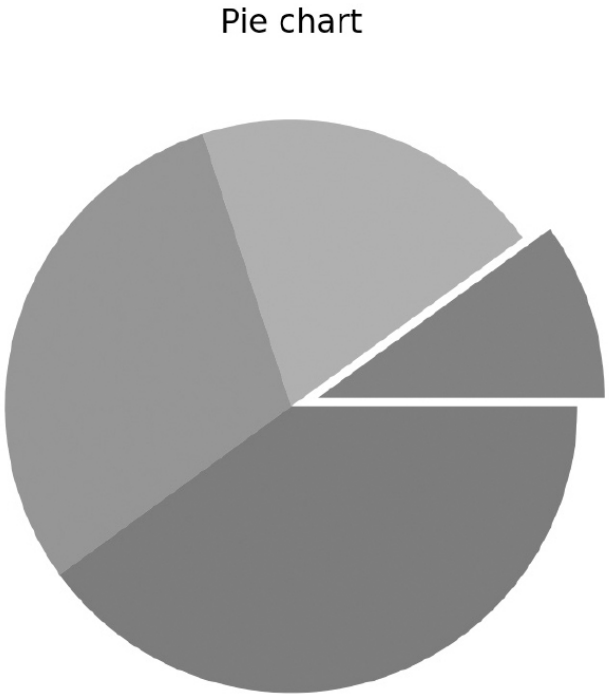


FIGURE 5.20 A basic pie chart

HEAT MAPS

Listing 5.24 displays the content of `heatmap1.py` that illustrates how to render a heat map based on random data values.

Listing 5.24: heatmap1.py

```
import numpy as np

data = np.random.random((16, 16))
plt.imshow(data, cmap='tab20_r',
interpolation='nearest')
plt.show()
```

Listing 5.24 contains an `import` statement, followed by the variable `data` that is initialized as a 16x16 matrix of random values. The next code snippet renders the heat map, and the final code snippet displays the heat map. Launch the code in Listing 5.24 and you will see the following output:

```
data.head():
year   1949  1950  1951  1952  1953  1954  1955  1956  1957  1958  1959  1960
month
Jan    112   115   145   171   196   204   242   284   315   340   360   417
Feb    118   126   150   180   196   188   233   277   301   318   342   391
Mar    132   141   178   193   236   235   267   317   356   362   406   419
Apr    129   135   163   181   235   227   269   313   348   348   396   461
May    121   125   172   183   229   234   270   318   355   363   420   472
```

In addition to the preceding data, you will also see the image that is shown in Figure 5.21.

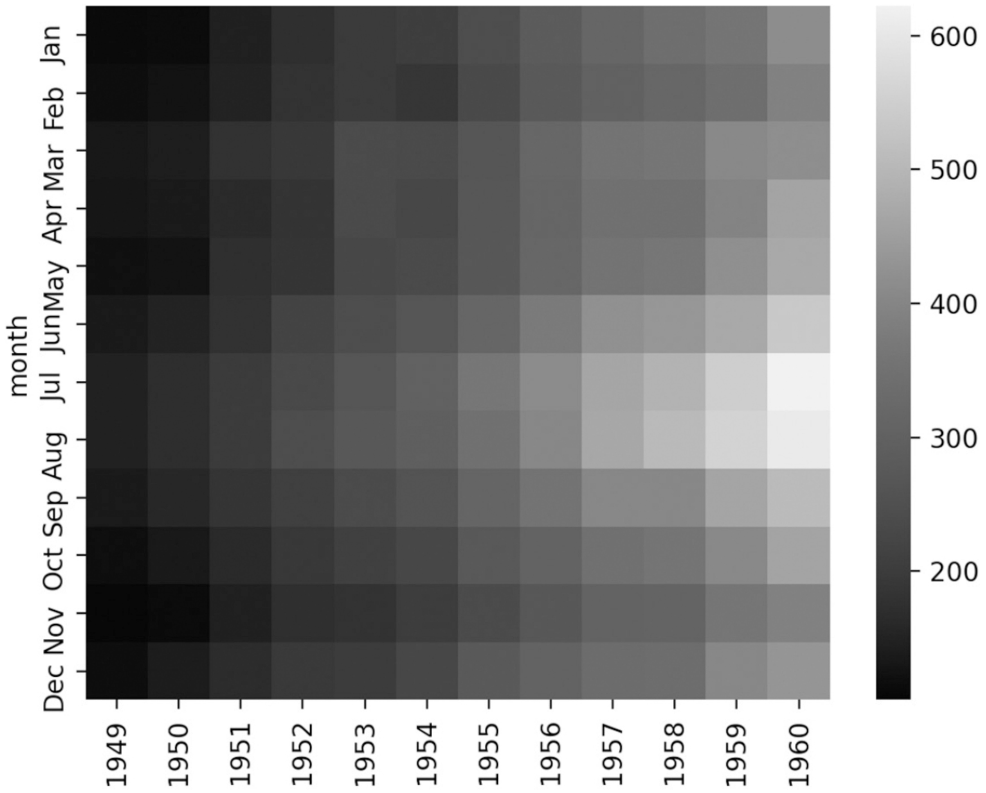


FIGURE 5.21 A heat map from random data

SAVE PLOT AS A PNG FILE

Listing 5.25 displays the content of `matplot2png.py` that shows you how to save a graphics image as a PNG file.

Listing 5.25: matplot2png.py

```
import matplotlib.pyplot as plt
import numpy as np

outfile="graph1.png"

plt.figure()
plt.plot(range(6))
```

```
fig, ax = plt.subplots()

ax.plot([2, 3, 4, 5, 5, 6, 6],
        [5, 7, 1, 3, 4, 6 ,8])

ax.plot([1, 2, 3, 4, 5],
        [2, 3, 4, 5, 6])

x = np.linspace(0, 12, 100)
plt.plot(np.sin(x))
plt.plot(np.linspace(-4,4,50))

plt.savefig(outfile, dpi=300)
```

Listing 5.25 contains `import` statements, followed by the variable `outfile` that is initialized with the name of the PNG file that will be saved to the file system. The contents of the PNG file consist of a sine wave and a set of line segments. Launch the code in Listing 5.25, and you will see the image that is shown in Figure 5.22.

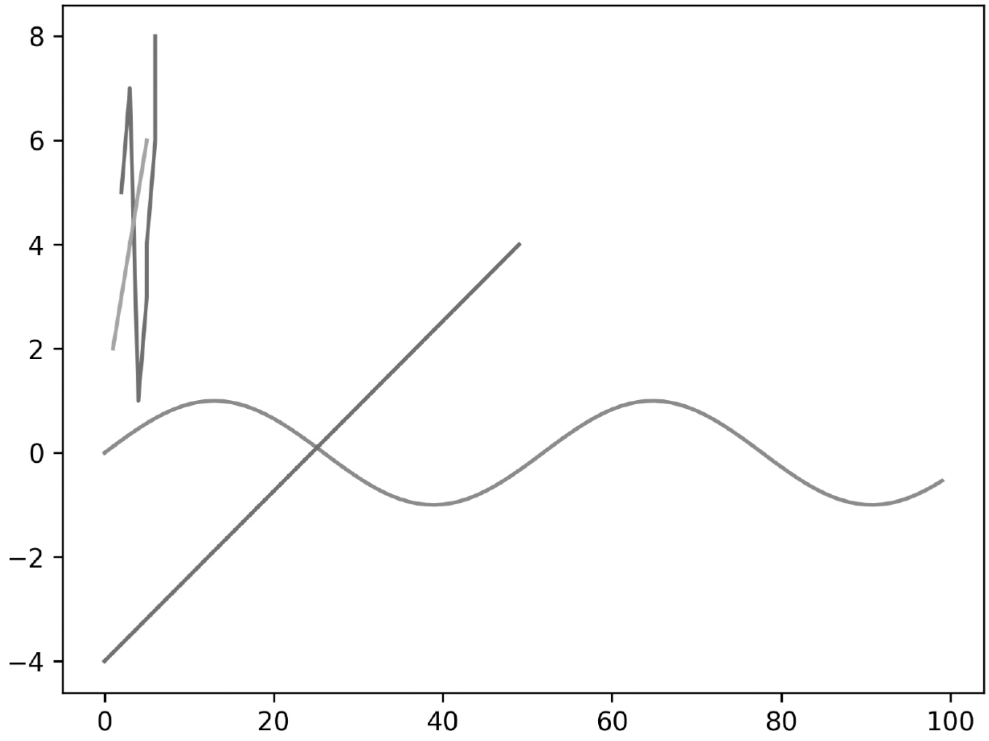


FIGURE 5.22 A random image

WORKING WITH SWEETVIZ

SweetViz is an open source Python module that generates remarkably detailed visualizations in the form of HTML webpages based on five lines of Python code.

As an illustration of the preceding statement, Listing 5.26 shows the content of `sweetviz1.py` that generates a visualization of various aspects of the `Iris` dataset that is available in scikit-learn.

Listing 5.26: sweetviz1.py

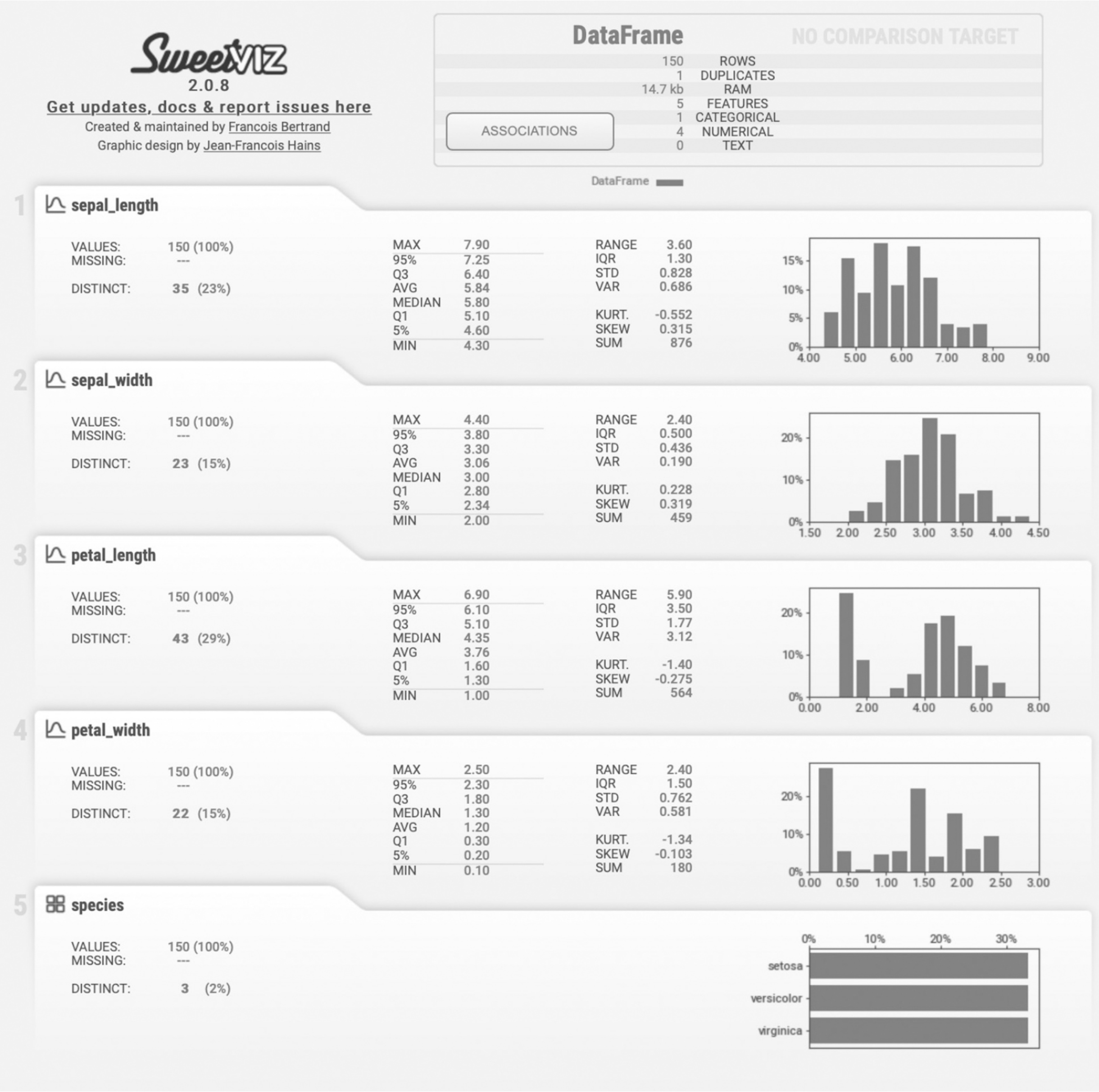
```
import sweetviz as sv
import seaborn as sns

df = sns.load_dataset('iris')
```

```
report = sv.analyze(df)
report.show_html()
```

Listing 5.26 starts with two `import` statements, followed an initialization of the variable `df` with the contents of the `Iris` dataset. The next code snippet initializes the variable report as the result of invoking the `analyze()` method in `SweetViz`, followed by a code snippet that generates an HTML webpage with the result of the analysis.

Launch the code from the command line, and you will see a new HTML webpage called `SWEETVIZ_REPORT.html` in the same directory. Figure 5.23 shows the content of the webpage `SWEETVIZ_REPORT.html`.


SweetVIZ
2.0.8
Get updates, docs & report issues here
ASSOCIATIONS
NO COMPARISON TARGET
DataFrame
15%
10%
5%
0%
4.00
5.00
6.00
7.00
8.00
9.00
20%
10%
0%
1.50
2.00
2.50
3.00
3.50
4.00
4.50
20%
10%
0%
0.00
2.00
4.00
6.00
8.00
20%
10%
0%
0.00
0.50
1.00
1.50
2.00
2.50
3.00
0%
10%
20%
30%
setosa
versicolor
virginica

Created & maintained by Francois Bertrand
Graphic design by Jean-Francois Hains

DataFrame

150 ROWS
1 DUPLICATES
14.7 kb RAM
5 FEATURES
1 CATEGORICAL
4 NUMERICAL
0 TEXT

1 sepal_length

VALUES: 150 (100%)
MISSING: ---
DISTINCT: 35 (23%)

MAX 7.90
95% 7.25
Q3 6.40
AVG 5.84
MEDIAN 5.80
Q1 5.10
5% 4.60
MIN 4.30

RANGE 3.60
IQR 1.30
STD 0.828
VAR 0.686
KURT. -0.552
SKEW 0.315
SUM 876

2 sepal_width

VALUES: 150 (100%)
MISSING: ---
DISTINCT: 23 (15%)

MAX 4.40
95% 3.80
Q3 3.30
AVG 3.06
MEDIAN 3.00
Q1 2.80
5% 2.34
MIN 2.00

RANGE 2.40
IQR 0.500
STD 0.436
VAR 0.190
KURT. 0.228
SKEW 0.319
SUM 459

3 petal_length

VALUES: 150 (100%)
MISSING: ---
DISTINCT: 43 (29%)

MAX 6.90
95% 6.10
Q3 5.10
MEDIAN 4.35
AVG 3.76
Q1 1.60
5% 1.30
MIN 1.00

RANGE 5.90
IQR 3.50
STD 1.77
VAR 3.12
KURT. -1.40
SKEW -0.275
SUM 564

4 petal_width

VALUES: 150 (100%)
MISSING: ---
DISTINCT: 22 (15%)

MAX 2.50
95% 2.30
Q3 1.80
MEDIAN 1.30
AVG 1.20
Q1 0.30
5% 0.20
MIN 0.10

RANGE 2.40
IQR 1.50
STD 0.762
VAR 0.581
KURT. -1.34
SKEW -0.103
SUM 180

5 species

VALUES: 150 (100%)
MISSING: ---
DISTINCT: 3 (2%)

FIGURE 5.23 An analysis of the Iris dataset

WORKING WITH SKIMPY

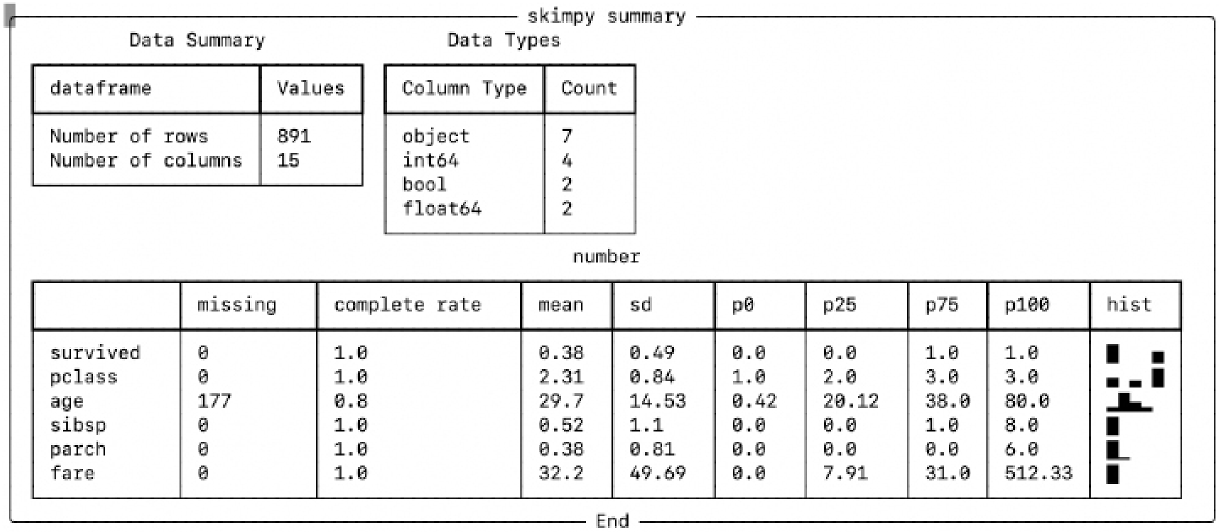
Skimpy is an open source Python module that generates an analysis of a dataset directly from the command line: no Python code is required. Install Skimpy with the following command:

```
pip3 install skimpy
```

Launch the following command to analyze the `Titanic` dataset (or a dataset of your own choice) that redirects the output to a text file (the latter is optional):

```
skimpy titanic.csv >titanic_out.txt
```

Figure 5.24 shows the generated output.



```
─────────────────────────────── skimpy summary ───────────────────────────────
          Data Summary                     Data Types
 ┌─────────────────────┬────────┐  ┌─────────────┬───────┐
 │ dataframe           │ Values │  │ Column Type │ Count │
 ├─────────────────────┼────────┤  ├─────────────┼───────┤
 │ Number of rows      │ 891    │  │ object      │ 7     │
 │ Number of columns   │ 15     │  │ int64       │ 4     │
 └─────────────────────┴────────┘  │ bool        │ 2     │
                                   │ float64     │ 2     │
                                   └─────────────┴───────┘
                                   number
 ┌──────────┬─────────┬───────────────┬──────┬───────┬──────┬───────┬──────┬────────┬──────┐
 │          │ missing │ complete rate │ mean │ sd    │ p0   │ p25   │ p75  │ p100   │ hist │
 ├──────────┼─────────┼───────────────┼──────┼───────┼──────┼───────┼──────┼────────┼──────┤
 │ survived │ 0       │ 1.0           │ 0.38 │ 0.49  │ 0.0  │ 0.0   │ 1.0  │ 1.0    │      │
 │ pclass   │ 0       │ 1.0           │ 2.31 │ 0.84  │ 1.0  │ 2.0   │ 3.0  │ 3.0    │      │
 │ age      │ 177     │ 0.8           │ 29.7 │ 14.53 │ 0.42 │ 20.12 │ 38.0 │ 80.0   │      │
 │ sibsp    │ 0       │ 1.0           │ 0.52 │ 1.1   │ 0.0  │ 0.0   │ 1.0  │ 8.0    │      │
 │ parch    │ 0       │ 1.0           │ 0.38 │ 0.81  │ 0.0  │ 0.0   │ 0.0  │ 6.0    │      │
 │ fare     │ 0       │ 1.0           │ 32.2 │ 49.69 │ 0.0  │ 7.91  │ 31.0 │ 512.33 │      │
 └──────────┴─────────┴───────────────┴──────┴───────┴──────┴───────┴──────┴────────┴──────┘
──────────────────────────────────── End ─────────────────────────────────────
```

FIGURE 5.24 An analysis of the Titanic dataset

3D CHARTS IN MATPLOTLIB

Listing 5.27 displays the content of `matplot_3d.py` that illustrates how to render a 3D plot in Matplotlib.

Listing 5.27: matplot_3d

```
import matplotlib.pyplot as plt
import numpy as np
```

```
zline = np.linspace(0,40,1000)
xline = 2*np.sin(2*zline)
yline = 3*np.cos(3*zline)
ax = plt.axes(projection="3d")
ax.plot3D(xline,yline,zline,'red',linewidth=4)
plt.show()
```

Listing 5.27 contains `import` statements, followed by the variables `zline`, `xline`, and `yline` that are initialized via the NumPy methods `linspace()`, `sin()`, and `cos()`, respectively. The next portion of Listing 5.28 initializes the variable `ax` to display a 3D effect, which is rendered by the final code snippet. Launch the code in Listing 5.27, and you will see the image that is shown in Figure 5.25.

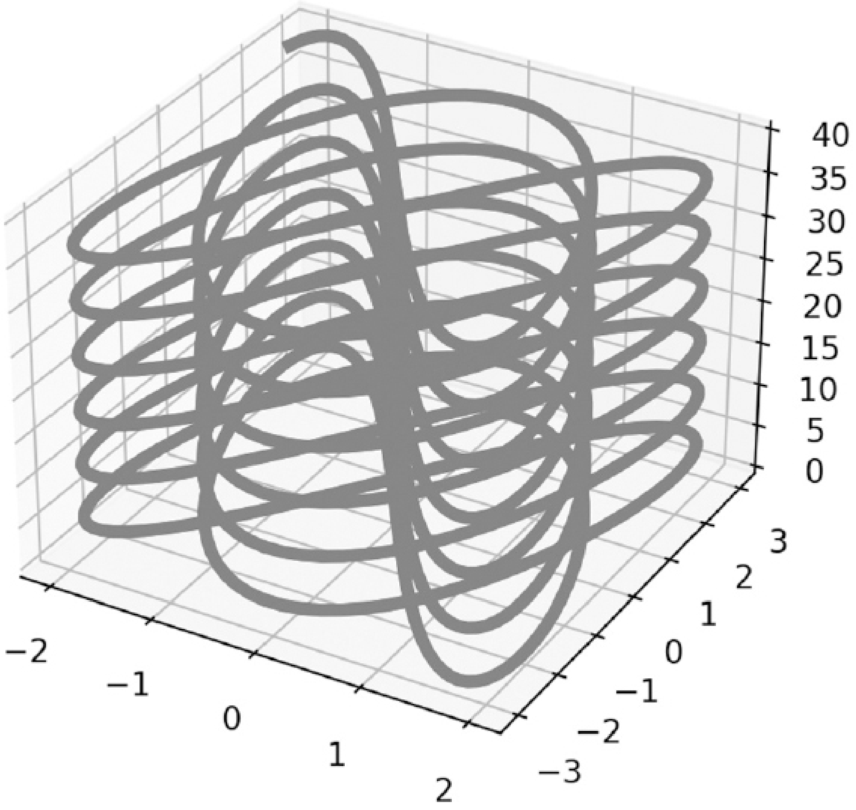


FIGURE 5.25 A 3D plot from trigonometric data

PLOTTING FINANCIAL DATA WITH MPLFINANCE

The section contains a Python-based code sample that shows you how to plot financial data for a given stock. First, make sure that you have the necessary Python library installed, as shown here:

```
pip3 install mplfinance
```

Listing 5.28 displays the content of `financial_mpl.py` that illustrates how to plot financial data in Matplotlib.

Listing 5.28: financial_mpl.py

```
import matplotlib.pyplot as plt
import pandas as pd

csvfile="aapl.csv"
daily =
pd.read_csv(csvfile,index_col=0,parse_dates=True)
daily.index.name = 'Date'

print("daily.head():")
print(daily.head())
print()

print("daily.tail():")
print(daily.tail())

import mplfinance as mpf
mpf.plot(daily)

#Plot types: ohlc, candle, line, renko, and pnf
```

Listing 5.28 contains `import` statements, followed by the variable `csvfile` that contains AAPL data for the years 2017 and 2018. Next, the variable `daily` is initialized with the contents of `aapl.csv`, followed by a block of code that prints the first five lines and the final five lines of data in `aapl.csv`.

The final code snippet invokes the `plot()` method of the class `mpf` (which is imported from `mplfinance`) to render a chart. Launch the code in Listing 5.28, and you will see the following output:

```
 daily.head():
                   Open        High  ...   Adj Close    Volume
 Date                                ...
 2017-01-03  115.800003  116.330002  ...  114.311760  28781900
 2017-01-04  115.849998  116.510002  ...  114.183815  21118100
 2017-01-05  115.919998  116.860001  ...  114.764473  22193600
 2017-01-06  116.779999  118.160004  ...  116.043915  31751900
 2017-01-09  117.949997  119.430000  ...  117.106812  33561900

 [5 rows x 6 columns]

 daily.tail():
                   Open        High  ...   Adj Close    Volume
 Date                                ...
 2018-01-12  176.179993  177.360001  ...  177.089996  25418100
 2018-01-16  177.899994  179.389999  ...  176.190002  29565900
 2018-01-17  176.149994  179.250000  ...  179.100006  34386800
 2018-01-18  179.369995  180.100006  ...  179.259995  31193400
 2018-01-19  178.610001  179.580002  ...  178.460007  31269600

[5 rows x 6 columns]
```

Figure 5.26 shows a plot of financial data based on the code in Listing 5.28.

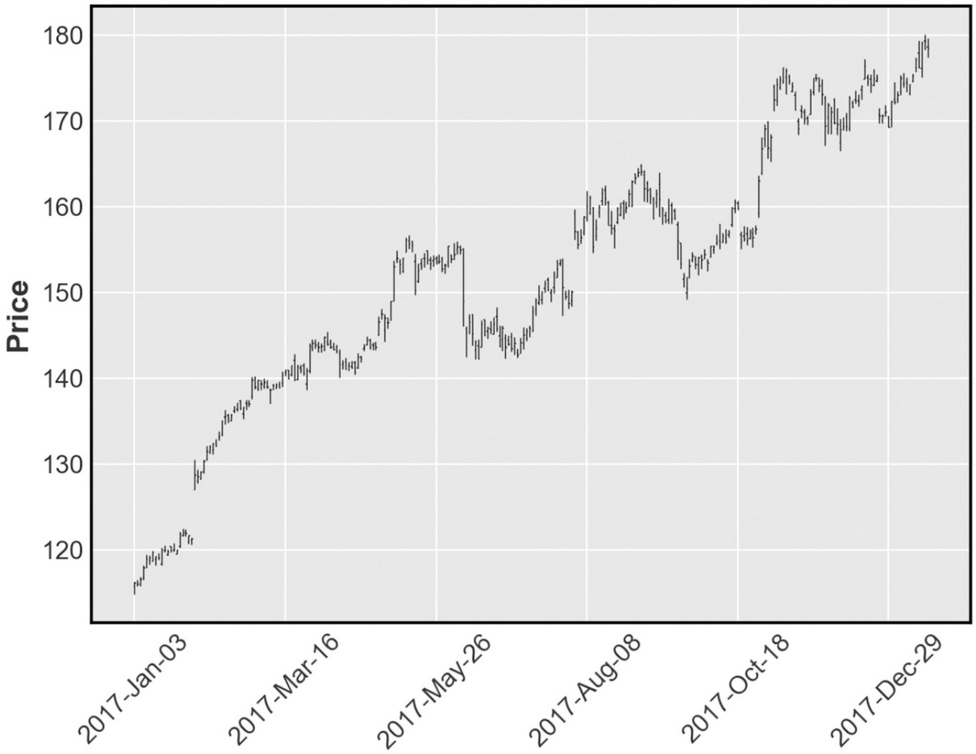


FIGURE 5.26 Plot of financial data

CHARTS AND GRAPHS WITH DATA FROM SQLITE3

Listing 5.29 displays the content of `rainfall_multiple.py` that shows you how to generate multiple charts and graphs from that that is extracted from a sqlite3 database.

Listing 5.29: rainfall_multiple.py

```
import sqlite3
import pandas as pd
import matplotlib.pyplot as plt

sql = """
  SELECT
  cast(centimeters/5.00 as int)*5 as cent_floor,
  count(*) as count
FROM rainfall
GROUP by 1
ORDER by 1;
"""

con = sqlite3.connect(" rain.db")
df = pd.read_sql_query(sql, con)
con.close()

####################################
# generate 7 types of charts/graphs
# and save them as PNG or TIFF files
####################################
df.hist(column='count', bins=14, grid=False,
rwidth=.8, color='red')
plt.savefig("rainfall_histogram.tiff")

df.plot.pie(y='count',figsize=(8,6))
plt.savefig("rainfall_pie.png")

df.plot.line(y='count',figsize=(8,6))
plt.savefig("rainfall_line.png")

df.plot.scatter(y='count',x='cent_floor',figsize=
(8,6))
plt.savefig("rainfall_scatter.png")

df.plot.box(figsize=(8,6))
plt.savefig("rainfall_box.png")
```

```
df.plot.hexbin(x='count',
y='cent_floor',gridsize=30, figsize=(8,6))
plt.savefig("rainfall_hexbin.png")
df["cent_floor"].plot.kde()
plt.savefig("rainfall_kde.png")

df["count"].hist()
df.plot.line(x='count', y='cent_floor', figsize=
(8,6))
df.plot.scatter(x='count', y='cent_floor',
figsize=(8,6))
df.plot.box(figsize=(8,6))
df.plot.hexbin(x='count',
y='cent_floor',gridsize=30, figsize=(8,6))
df.plot.pie(y='cost', figsize=(8, 6))
df["cent_floor"].plot.kde()
```

Listing 5.29 contains several `import` statements and initializes the variable `sql` with a SQL statement that selects data from the `rainfall` table. The next portion of Listing 5.29 initializes the variable `con` for accessing the `rain.db` database, and then populates the Pandas data frame `df` with the result of executing the SQL statement contained in the variable `sql`.

The next portion of Listing 5.29 contains pairs of code snippets for rendering charts and graphs of the type histogram, pie, line, scatter, box, hexbin, and kde (kernel density estimation), respectively. Launch the code in Listing 5.29, and you will see the following output:

```
=> Histogram of Rainfall:
    cent_floor count
0            0    14
```

```
1            5    13
2           10     4
3           15    10
4           20     3
5           25     6
6           30     3
7           35     6
8           40     2
9           45     1
10          50     1
11          55     1
12          60     1
```

In addition to the preceding output, you will see the following files in the same directory where you launched Listing 5.29:

```
rainfall_histogram.tiff
rainfall_pie.png
rainfall_line.png
rainfall_scatter.png
rainfall_box.png
rainfall_hexbin.png
rainfall_kde.png
```

WORKING WITH SEABORN

Seaborn is a Python library for data visualization that also provides a high-level interface to Matplotlib. Seaborn is easier to work with than Matplotlib, and actually extends Matplotlib, but Seaborn is not as powerful as Matplotlib.

Seaborn addresses two challenges of Matplotlib. The first involves the default Matplotlib parameters. Seaborn works with different parameters, which provides greater flexibility than the default rendering of Matplotlib plots. Seaborn addresses the limitations of the Matplotlib default values for features such as colors, tick marks on the upper and right axes, and the style (among others).

In addition, Seaborn makes it easier to plot entire data frames (somewhat like Pandas) than doing so in Matplotlib. Nevertheless, since Seaborn extends Matplotlib, knowledge of the latter (discussed in Chapter 6) is advantageous and will simplify your learning curve.

Features of Seaborn

Seaborn provides a nice set of features and useful methods to control the display of data, some of which are listed here:

- scale Seaborn plots
- set the plot style
- set the figure size
- rotate label text
- set xlim or ylim
- set log scale
- add titles

Some useful Seaborn methods are listed here:

- `plt.xlabel()`
- `plt.ylabel()`
- `plt.annotate()`
- `plt.legend()`
- `plt.ylim()`
- `plt.savefig()`

Seaborn supports various built-in datasets, just like NumPy and Pandas, including the Iris dataset and the Titanic dataset, both of which you will see in subsequent sections. As a starting point, the next section contains the code that displays all the available built-in datasets in Seaborn.

SEABORN DATASET NAMES

Listing 5.30 displays the content `dataset_names.py` that shows the Seaborn built-in datasets, one of which we will use in a subsequent section to render a heat map in Seaborn.

Listing 5.30: dataset_names.py

```
import seaborn as sns

names = sns.get_dataset_names()
for name in names:
  print("name:",name)
```

Listing 5.30 contains an `import` statement, followed by initializing the variable `names` with a list of the dataset names in Seaborn. Then a simple loop iterates through the values in the `names` variable and prints their values. The output from Listing 5.30 is here:

```
name: anagrams
name: anscombe
name: attention
name: brain_networks
name: car_crashes
name: diamonds
name: dots
```

```
name: exercise
name: flights
name: fmri
name: gammas
name: geyser
name: iris
name: mpg
name: penguins
name: planets
name: taxis
name: tips
name: titanic
```

The three-line code sample in the next section shows you how to display the rows in the built-in “tips” dataset.

SEABORN BUILT-IN DATASETS

Listing 5.31 displays the content of `seaborn_tips.py` that illustrates how to read the `tips` dataset into a data frame and display the first five rows of the dataset.

Listing 5.31: seaborn_tips.py

```
import seaborn as sns

df = sns.load_dataset("tips")
print(df.head())
```

Listing 5.31 is very simple: after importing Seaborn, the variable `df` is initialized with the data in the built-in dataset `tips`, and the `print()`

statement displays the first five rows of `df`. Note that the `load_dataset()` API searches for online or built-in datasets. The output from Listing 5.31 is here:

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

THE IRIS DATASET IN SEABORN

Listing 5.32 displays the content of `seaborn_iris.py` that illustrates how to plot the `Iris` dataset.

Listing 5.32: seaborn_iris.py

```
import seaborn as sns
import Matplotlib.pyplot as plt

# Load iris data
iris = sns.load_dataset("iris")

# Construct iris plot
sns.swarmplot(x="species", y="petal_length",
data=iris)

# Show plot
plt.show()
```

Listing 5.32 imports `seaborn` and `Matplotlib.pyplot` and then initializes the variable `iris` with the contents of the built-in `Iris` dataset. Next, the `swarmplot()` API displays a graph with the horizontal axis labeled `species`, the vertical axis labeled `petal_length`, and the displayed points are from the `Iris` dataset.

Figure 5.27 shows the images in the `Iris` dataset based on the code in Listing 5.32.

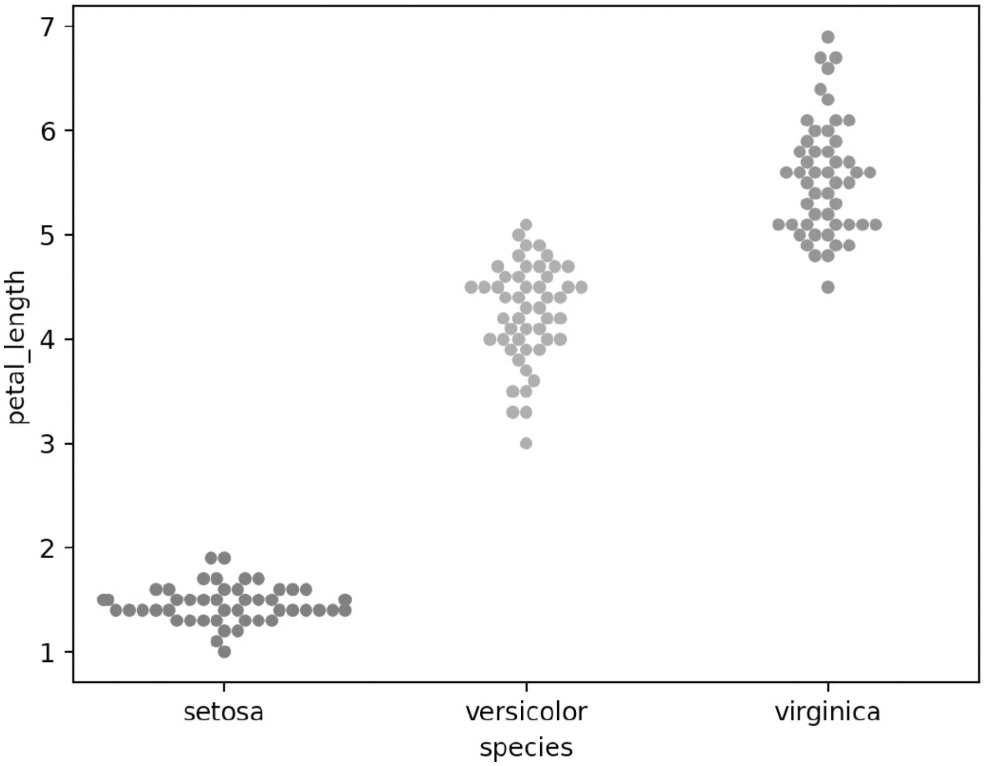


FIGURE 5.27 The Iris dataset

THE TITANIC DATASET IN SEABORN

Listing 5.33 displays the content of `seaborn_titanic_plot.py` that illustrates how to plot the `Titanic` dataset.

Listing 5.33: seaborn_titanic_plot.py

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic = sns.load_dataset("titanic")
g = sns.factorplot("class", "survived", "sex",
data=titanic, kind="bar", palette="muted",
legend=False)

plt.show()
```

Listing 5.33 contains the same `import` statements as Listing 5.33, and then initializes the variable `titanic` with the contents of the built-in `Titanic` dataset. Next, the `factorplot()` API displays a graph with dataset attributes that are listed in the API invocation.

Figure 5.28 shows a plot of the data in the `Titanic` dataset based on the code in Listing 5.33.

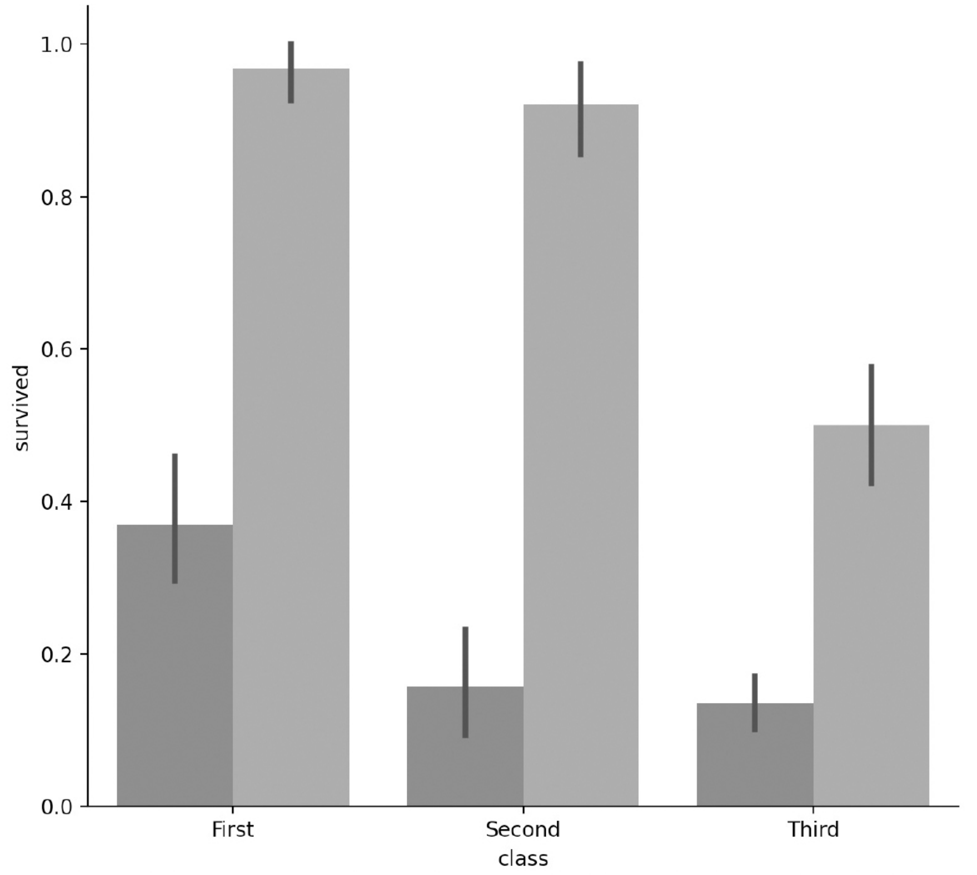


FIGURE 5.28 A histogram of the Titanic dataset.

EXTRACTING DATA FROM TITANIC DATASET IN SEABORN (1)

Listing 5.34 displays the content of `seaborn_titanic.py` that illustrates how to extract subsets of data from the `Titanic` dataset.

Listing 5.34: seaborn_titanic.py

```
import matplotlib.pyplot as plt
import seaborn as sns
titanic = sns.load_dataset("titanic")
print("titanic info:")
```

```
titanic.info()

print("first five rows of titanic:")
print(titanic.head())

print("first four ages:")
print(titanic.loc[0:3,'age'])

print("fifth passenger:")
print(titanic.iloc[4])

#print("first five ages:")
#print(titanic['age'].head())

#print("first five ages and gender:")
#print(titanic[['age','sex']].head())

#print("descending ages:")
#print(titanic.sort_values('age', ascending =
False).head())

#print("older than 50:")
#print(titanic[titanic['age'] > 50])

#print("embarked (unique):")
#print(titanic['embarked'].unique())

#print("survivor counts:")
#print(titanic['survived'].value_counts())

#print("counts per class:")
#print(titanic['pclass'].value_counts())

#print("max/min/mean/median ages:")
```

```
#print(titanic['age'].max())
#print(titanic['age'].min())
#print(titanic['age'].mean())
#print(titanic['age'].median())
```

Listing 5.34 contains the same `import` statements as Listing 5.34, and then initializes the variable `titanic` with the contents of the built-in `Titanic` dataset. The next portion of Listing 5.34 displays various aspects of the `Titanic` dataset, such as its structure, the first five rows, the first four ages, and the details of the fifth passenger.

As you can see, there is a large block of “commented out” code that you can uncomment to see the associated output, such as age, gender, persons over 50, unique rows, and so forth. The output from Listing 5.34 is here:

```
#print(titanic['age'].mean())
titanic info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
survived     891 non-null int64
pclass       891 non-null int64
sex          891 non-null object
age          714 non-null float64
sibsp        891 non-null int64
parch        891 non-null int64
fare         891 non-null float64
embarked     889 non-null object
class        891 non-null category
who          891 non-null object
adult_male   891 non-null bool
```

```
deck         203 non-null category
embark_town  889 non-null object
alive        891 non-null object
alone        891 non-null bool
dtypes: bool(2), category(2), float64(2),
int64(4), object(5)
memory usage: 80.6+ KB
first five rows of titanic:

   survived pclass sex    age  sibsp parch fare    embarked class \
0         0      3 male   22.0 1     0     7.2500  S        Third
1         1      1 female 38.0 1     0     71.2833 C        First
2         1      3 female 26.0 0     0     7.9250  S        Third
3         1      1 female 35.0 1     0     53.1000 S        First
4         0      3 male   35.0 0     0     8.0500  S        Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
first four ages:
0   22.0
1   38.0
2   26.0
3   35.0
Name: age, dtype: float64
fifth passenger:
survived                0
pclass                  3
sex                  male
```

```
age                     35
sibsp                    0
parch                    0
fare                  8.05
embarked                 S
class                Third
who                    man
adult_male            True
deck                   NaN
embark_town    Southampton
alive                   no
alone                 True
Name: 4, dtype: object
counts per class:
3    491
1    216
2    184
Name: pclass, dtype: int64
max/min/mean/median ages:
80.0
0.42
29.69911764705882
28.0
```

EXTRACTING DATA FROM TITANIC DATASET IN SEABORN (2)

Listing 5.35 displays the content of `seaborn_titanic2.py` that illustrates how to extract subsets of data from the `Titanic` dataset.

Listing 5.35: seaborn_titanic2.py

```
import matplotlib.pyplot as plt
import seaborn as sns
titanic = sns.load_dataset("titanic")

# Returns a scalar
# titanic.ix[4, 'age']
print("age:", titanic.at[4, 'age'])

# Returns a Series of name 'age', and the age
values associated
# to the index labels 4 and 5
# titanic.ix [[4, 5], 'age']
print("series:",titanic.loc[[4, 5], 'age'])

# Returns a Series of name '4', and the age and
fare values
# associated to that row.
# titanic.ix [4, ['age', 'fare']]
print("series:",titanic.loc[4, ['age', 'fare']])

# Returns a DataFrame with rows 4 and 5, and
columns 'age' and 'fare'
# titanic.ix [[4, 5], ['age', 'fare']]
print("dataframe:",titanic.loc[[4, 5], ['age',
'fare']])

query = titanic[
    (titanic.sex == 'female')
    & (titanic['class'].isin(['First', 'Third']))
    & (titanic.age > 30)
    & (titanic.survived == 0)
]
print("query:",query)
```

Listing 5.35 contains the same `import` statements as Listing 5.34, and then initializes the variable `titanic` with the contents of the built-in `Titanic` dataset. The next code snippet displays the age of the passenger with index 4 in the dataset (which equals 35).

The following code snippet displays the ages of passengers with index values 4 and 5 in the dataset:

```
print("series:",titanic.loc[[4, 5], 'age'])
```

The next snippet displays the age and fare of the passenger with index 4 in the dataset, followed by another code snippet displaying the age and fare of the passengers with index 4 and index 5 in the dataset.

The final portion of Listing 5.35 is the most interesting part: it defines a variable `query` as shown here:

```
query = titanic[
    (titanic.sex == 'female')
    & (titanic['class'].isin(['First', 'Third']))
    & (titanic.age > 30)
    & (titanic.survived == 0)
]
```

The preceding code block will retrieve the female passengers who are in either first class or third class, are over 30 years old, and did not survive the accident. The entire output from Listing 5.35 is here:

```
age: 35.0
series: 4    35.0
5      NaN
Name: age, dtype: float64
series: age       35
fare    8.05
Name: 4, dtype: object
dataframe:      age    fare
4  35.0  8.0500
5   NaN  8.4583

query:  survived  pclass     sex   age  sibsp  parch    fare embarked
class  \
18          0       3  female  31.0      1      0  18.0000        S  Third
40          0       3  female  40.0      1      0   9.4750        S  Third
132         0       3  female  47.0      1      0  14.5000        S  Third
167         0       3  female  45.0      1      4  27.9000        S  Third
177         0       1  female  50.0      0      0  28.7125        C  First
254         0       3  female  41.0      0      2  20.2125        S  Third
276         0       3  female  45.0      0      0   7.7500        S  Third
362         0       3  female  45.0      0      1  14.4542        C  Third
396         0       3  female  31.0      0      0   7.8542        S  Third
503         0       3  female  37.0      0      0   9.5875        S  Third
610         0       3  female  39.0      1      5  31.2750        S  Third
638         0       3  female  41.0      0      5  39.6875        S  Third
657         0       3  female  32.0      1      1  15.5000        Q  Third
678         0       3  female  43.0      1      6  46.9000        S  Third
736         0       3  female  48.0      1      3  34.3750        S  Third
767         0       3  female  30.5      0      0   7.7500        Q  Third
885         0       3  female  39.0      0      5  29.1250        Q  Third
```

VISUALIZING A PANDAS DATA FRAME IN SEABORN

Listing 5.36 displays the content of `pandas_seaborn.py` that illustrates how to display a Pandas dataset in Seaborn.

Listing 5.36: pandas_seaborn.py

```
import pandas as pd
import random
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.DataFrame()
df['x'] = random.sample(range(1, 100), 25)
df['y'] = random.sample(range(1, 100), 25)

print("top five elements:")
print(df.head())

# display a density plot
#sns.kdeplot(df.y)

# display a density plot
#sns.kdeplot(df.y, df.x)

#sns.distplot(df.x)

# display a histogram
#plt.hist(df.x, alpha=.3)
#sns.rugplot(df.x)

# display a boxplot
```

```
#sns.boxplot([df.y, df.x])

# display a violin plot
#sns.violinplot([df.y, df.x])

# display a heatmap
#sns.heatmap([df.y, df.x], annot=True, fmt="d")

# display a cluster map
#sns.clustermap(df)

# display a scatterplot of the data points
sns.lmplot('x', 'y', data=df, fit_reg=False)
plt.show()
```

Listing 5.36 contains several familiar `import` statements, followed by the initialization of the Pandas variable `df` as a Pandas data frame. The next two code snippets initialize the columns and rows of the data frame, and the `print()` statement displays the first five rows.

For your convenience, Listing 5.37 contains an assortment of “commented out” code snippets that use Seaborn to render a density plot, a histogram, a boxplot, a violin plot, a heatmap, and a cluster. Uncomment the portions that interest you to see the associated plot. The output from Listing 5.37 is here:

```
top five elements:
    x  y
0  52 34
1  31 47
2  23 18
3  34 70
4  71  1
```

Figure 5.29 shows a plot of the data in the `Titanic` dataset based on the code in Listing 5.36.

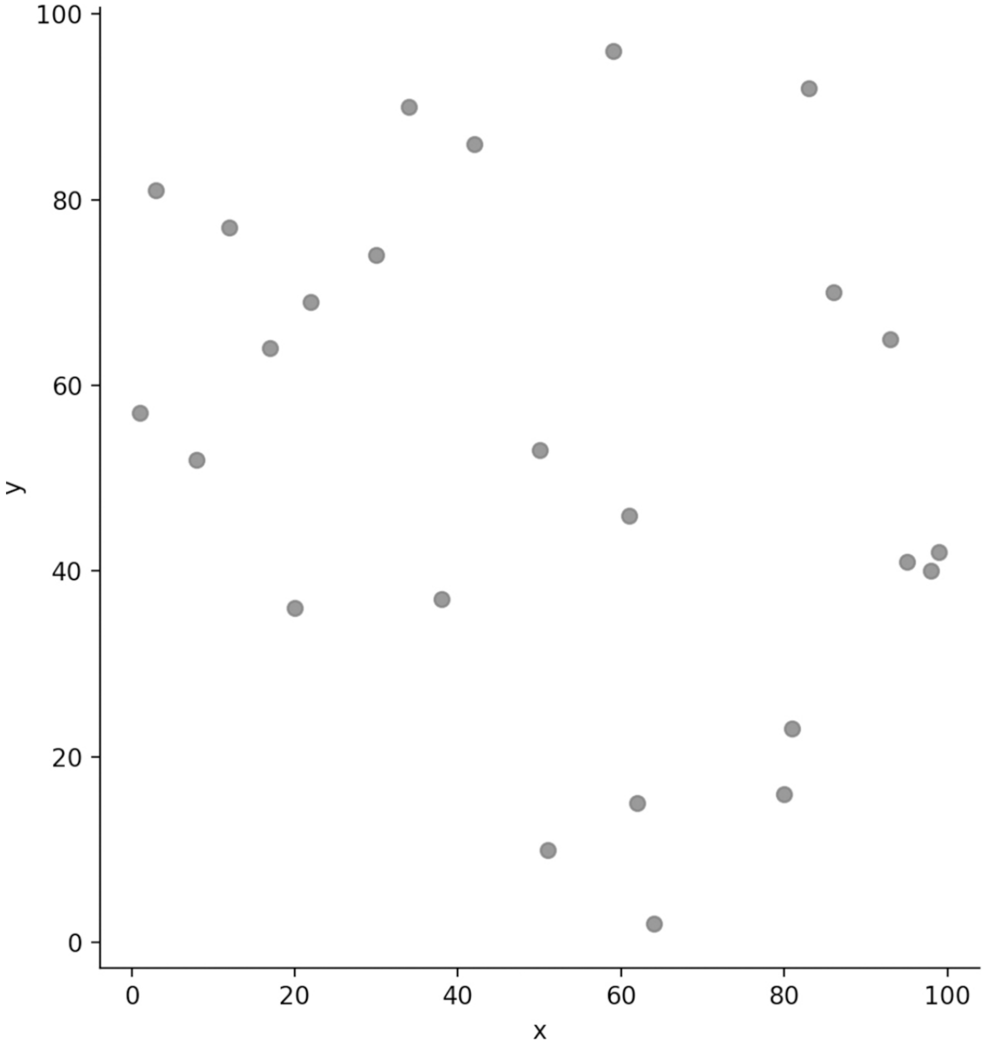


FIGURE 5.29 A Pandas data frame displayed via Seaborn

SEABORN HEAT MAPS

Listing 5.37 displays the contents `sns_heatmap1.py` that shows a heat map from a Seaborn built-in dataset.

Listing 5.37: sns_heatmap1.py

```
import seaborn as sns
import matplotlib.pyplot as plt

data = sns.load_dataset("flights")
data = data.pivot("month", "year", "passengers")

print("data.head():")
print(data.head())

sns.heatmap(data)
plt.show()
```

Listing 5.37 contains `import` statements and then initializes the variable `data` with the built-in `flights` dataset. The next code snippet invokes the `pivot()` method that “inverts” the row and columns of the dataset. The final code portion of Listing 5.37 displays the first five rows of the dataset and then generates a heat map based on the dataset. The output from Listing 5.37 is here:

```
data.head():
year   1949  1950  1951  1952  1953  1954  1955  1956  1957  1958  1959  1960
month
Jan     112   115   145   171   196   204   242   284   315   340   360   417
Feb     118   126   150   180   196   188   233   277   301   318   342   391
Mar     132   141   178   193   236   235   267   317   356   362   406   419
Apr     129   135   163   181   235   227   269   313   348   348   396   461
May     121   125   172   183   229   234   270   318   355   363   420   472
```

Figure 5.30 shows a plot of the data in the `Titanic` dataset based on the code in Listing 5.37.

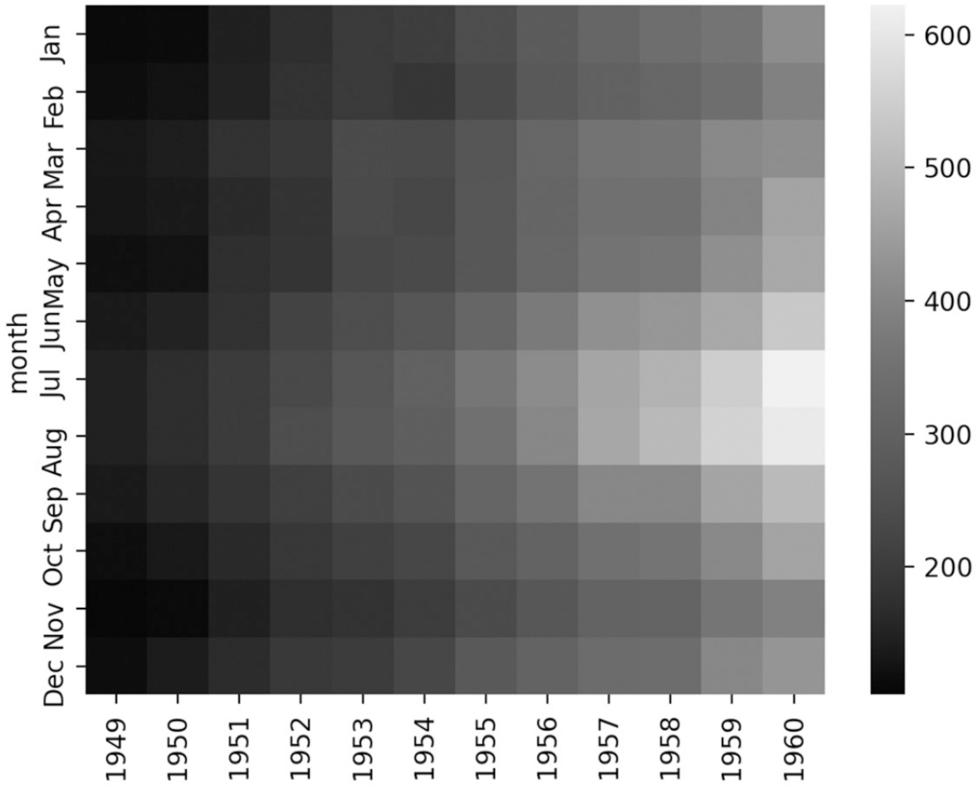


FIGURE 5.30 A Pandas data frame displayed via Seaborn

SEABORN PAIR PLOTS

This section contains several Python-based code samples that show you how to use the Seaborn `pair_plot()` method to render pair plots.

Listing 5.38 displays the contents `seaborn_pairplot1.py` that shows a pair plot with the `Iris` dataset.

Listing 5.38: seaborn_pairplot1.py

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

# load iris data
```

```
iris = sns.load_dataset("iris")

df = pd.DataFrame(iris)

# construct and display iris plot
g = sns.pairplot(df, height=2, aspect=1.0)
plt.show()
```

Listing 5.38 contains `import` statements and then initializes the variable `iris` with the built-in `iris` dataset. The next code snippet initializes the data frame `df` with the contents of the `iris` dataset. The final code portion of Listing 5.38 constructs a pair plot of the `iris` dataset and then displays the output. Figure 5.31 shows a plot of the data in the `Titanic` dataset based on the code in Listing 5.38.

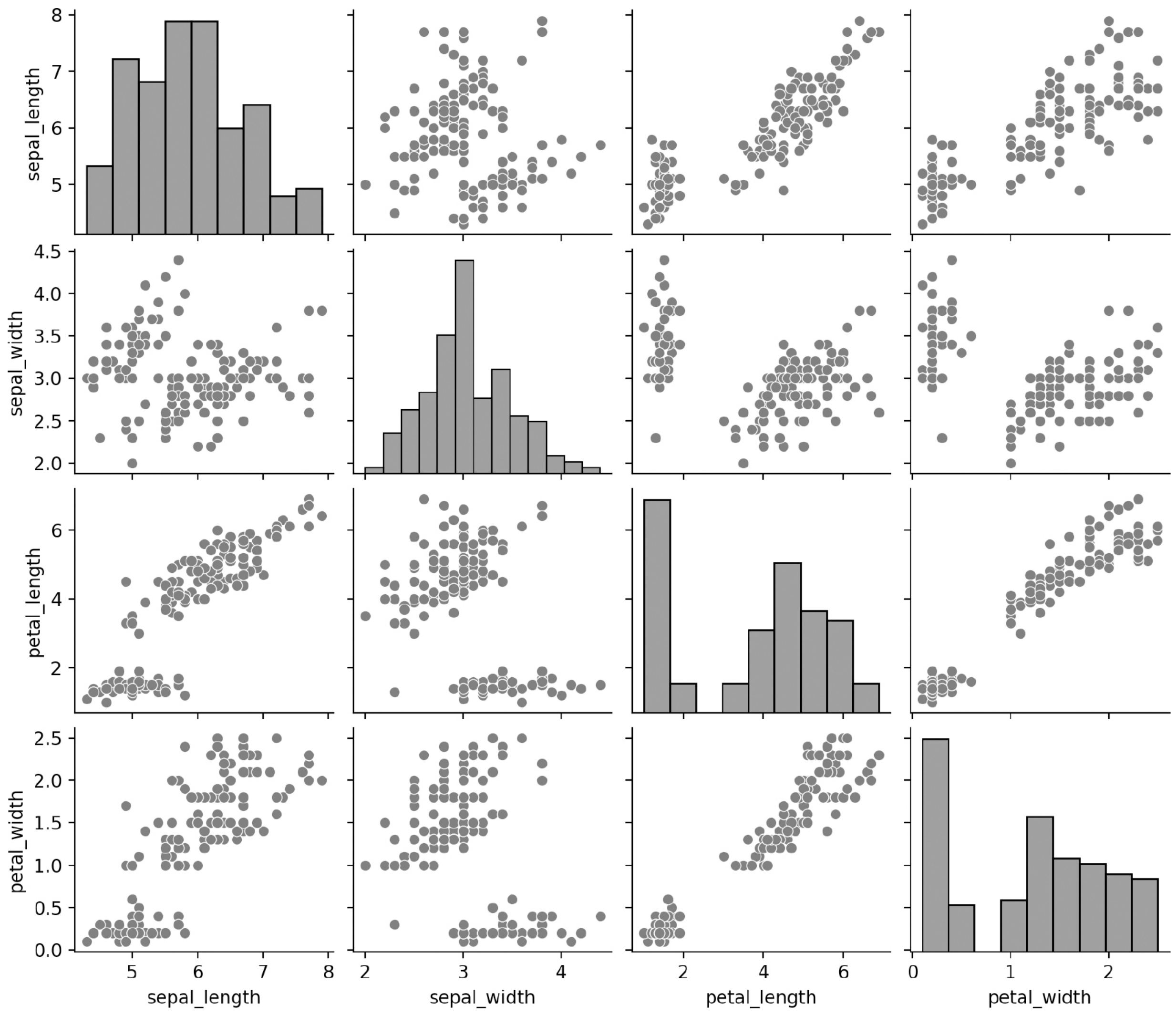


FIGURE 5.31 A Seaborn pair plot

Listing 5.39 displays the content `seaborn_pairplot2.py` that shows a pair plot with the `Iris` dataset.

Listing 5.39: seaborn_pairplot2.py

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

# load iris data
```

```
iris = sns.load_dataset("iris")

df = pd.DataFrame(iris)

# IRIS columns:
#
sepal_length,sepal_width,petal_length,petal_width,
species

# plot a subset of columns:
plot_columns = ['sepal_length', 'sepal_width']
sns.pairplot(df[plot_columns])
plt.show()

# specify KDE for the diagonal:
sns.pairplot(df[plot_columns], diag_kind='kde')
plt.show()
```

Listing 5.39 is similar to the code in Listing 5.38: the difference is that the former selects only two features in the `iris` dataset instead of all the features in the `iris` dataset.

The next code portion of Listing 5.39 construct a pair plot of the `iris` dataset and then displays the output, followed by another pair plot that specifies the `kde` value for the `diag_kind` parameter. More information about kde is discussed in the Seaborn documentation here:

https://seaborn.pydata.org/tutorial/distributions.html#tutorial-kde

Launch the code in Listing 5.39, and you will see a pair plot as shown in Figure 5.32.

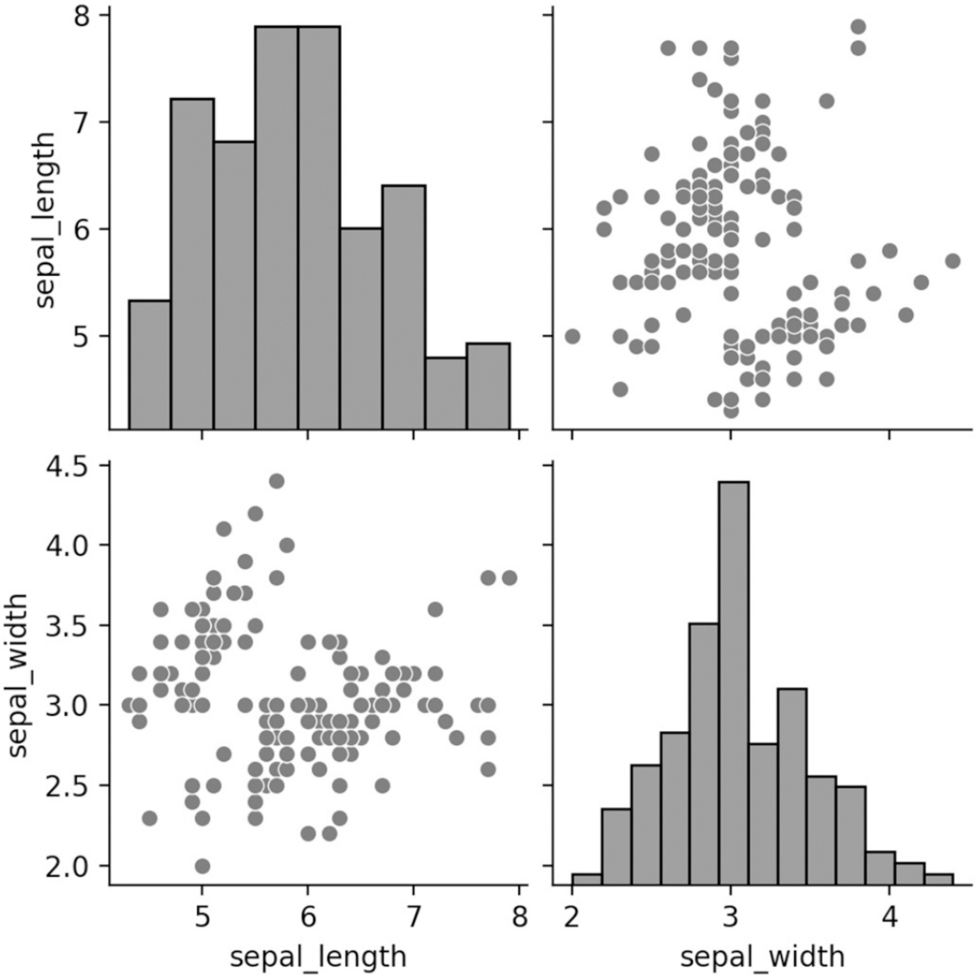


FIGURE 5.32 A Seaborn pair plot

Figure 5.33 displays a plot of the data with the `kde` option for the `iris` dataset based on the code in Listing 5.39.

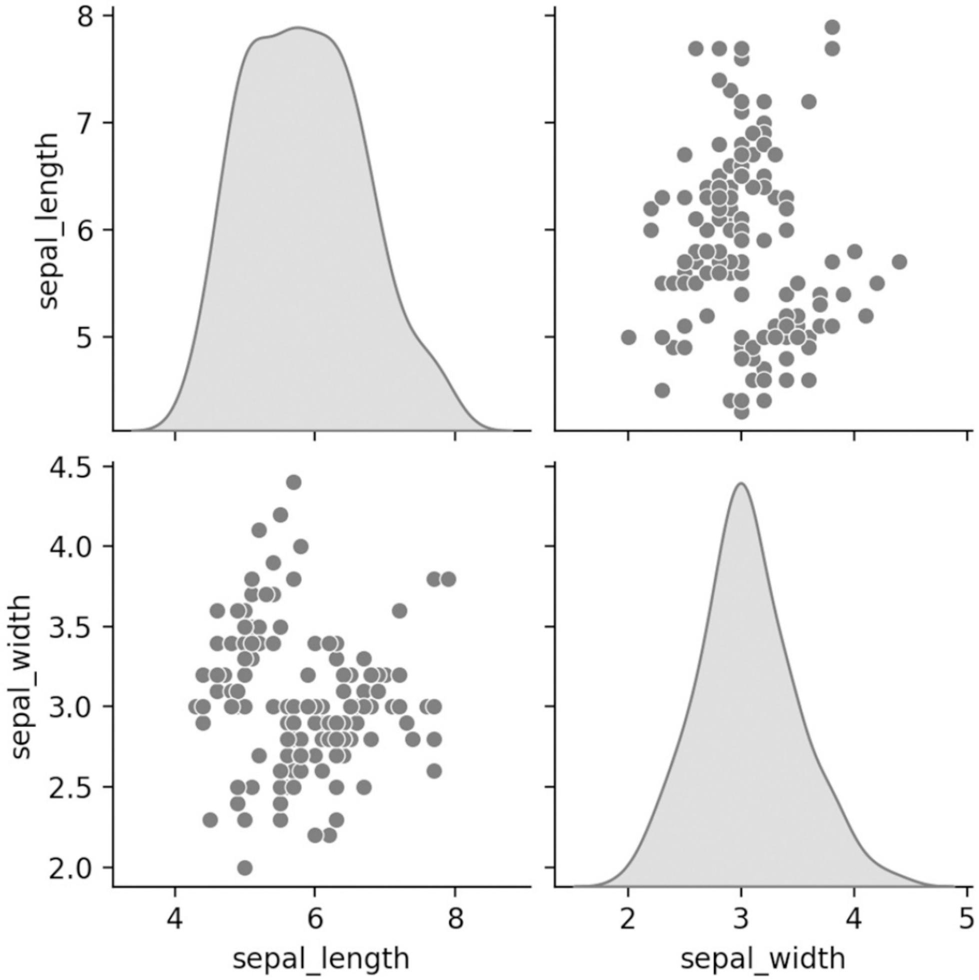


FIGURE 5.33 A Seaborn pair plot with kde

WHAT IS BOKEH?

Bokeh is an open source project that depends on Matplotlib as well as scikit-learn. As you will see in the subsequent code sample, Bokeh generates an HTML webpage that is based on Python code, and then launches that webpage in a browser. Bokeh and `D3.js` (which is a JavaScript layer of abstraction over SVG) both provide elegant visualization effects that support animation effects and user interaction.

Bokeh enables the rapid creation statistical visualization, and it works with other tools with as Python Flask and Django. In addition to Python, Bokeh supports Julia, Lua, and R (JSON files are generated instead of HTML webpages).

Listing 5.40 displays the content `bokeh_trig.py` that illustrates how to create a graphics effect using various Bokeh APIs.

Listing 5.40: bokeh_trig.py

```
# pip3 install bokeh
from bokeh.plotting import figure, output_file,
show
from bokeh.layouts import column
import bokeh.colors as colors
import numpy as np
import math

deltaY = 0.01
maxCount = 150
width = 800
height = 400
band_width = maxCount/3

x = np.arange(0, math.pi *3, 0.05)
y1 = np.sin(x)
y2 = np.cos(x)

white = colors.RGB(255,255,255)

fig1 = figure(plot_width = width, plot_height =
height)

for i in range(0,maxCount):
  rgb1 = colors.RGB(i*255/maxCount, 0, 0)
  rgb2 = colors.RGB(i*255/maxCount, i*255/maxCount,
  0)
  fig1.line(x, y1-i*deltaY,line_width = 2,
line_color = rgb1)
  fig1.line(x, y2-i*deltaY,line_width = 2,
line_color = rgb2)

for i in range(0,maxCount):
  rgb1 = colors.RGB(0, 0, i*255/maxCount)
  rgb2 = colors.RGB(0, i*255/maxCount, 0)
  fig1.line(x, y1+i*deltaY,line_width = 2,
line_color = rgb1)
  fig1.line(x, y2+i*deltaY,line_width = 2,
line_color = rgb2)
  if (i % band_width == 0):
    fig1.line(x, y1+i*deltaY,line_width = 5,
line_color = white)

show(fig1)
```

Listing 5.40 starts with a commented out `pip3` code snippet that you can launch from the command line to install Bokeh (in case you have not done so already).

The next code block contains several Bokeh-related statements, as well as NumPy and Math.

Notice that the variable `white` is defined as an (R, G, B) triple of integers, which represents the red, green, and blue components of a color. In particular, (255, 255, 255) represents the color white (check online if you are unfamiliar with RGB). The next portion of Listing 5.40 initializes some scalar variables that are used in the two `for` loops that are in the second half of Listing 5.40.

Next, the NumPy variable `x` is a range of values from 0 to `math.PI/3`, with an increment of 0.05 between successive values. Then the NumPy variables `y1` and `y2` are defined as the sine and cosine values, respectively, of the values in `x`. The next code snippet initializes the variable `fig1` that represents a context in which the graphics effects will be rendered. This completes the initialization of the variables that are used in the two for loops.

The next portion of Listing 5.40 contains the first `for` loop that creates a gradient-like effect by defining (R, G, B) triples whose values are based partially on the value of the loop variable `i`. For example, the variable `rgb1` ranges in a linear fashion from (0, 0, 0) to (255, 0, 0), which represent the colors black and red, respectively. The variable `rgb2` ranges in a linear fashion from (0, 0, 0) to (255, 255, 0), which represent the colors black and yellow, respectively. The next portion of the `for` loop contains two invocations of the `fig1.line()` API that renders a sine wave and a cosine wave in the context variable `fig1`.

The second loop is similar to the first loop: the main difference is that the variable `rgb1` varies from black to blue, and the variable `rgb2` variables from black to green. The final code snippet in Listing 5.40 invokes the `show()` method that generates an HTML webpage (with the same prefix as the Python file) and then launches the webpage in a browser.

Figure 5.34 displays the graphics effect based on the code in Listing 5.40. If this image is displayed as black and white, launch the code from the command line and you will see the gradient-like effects in the image.

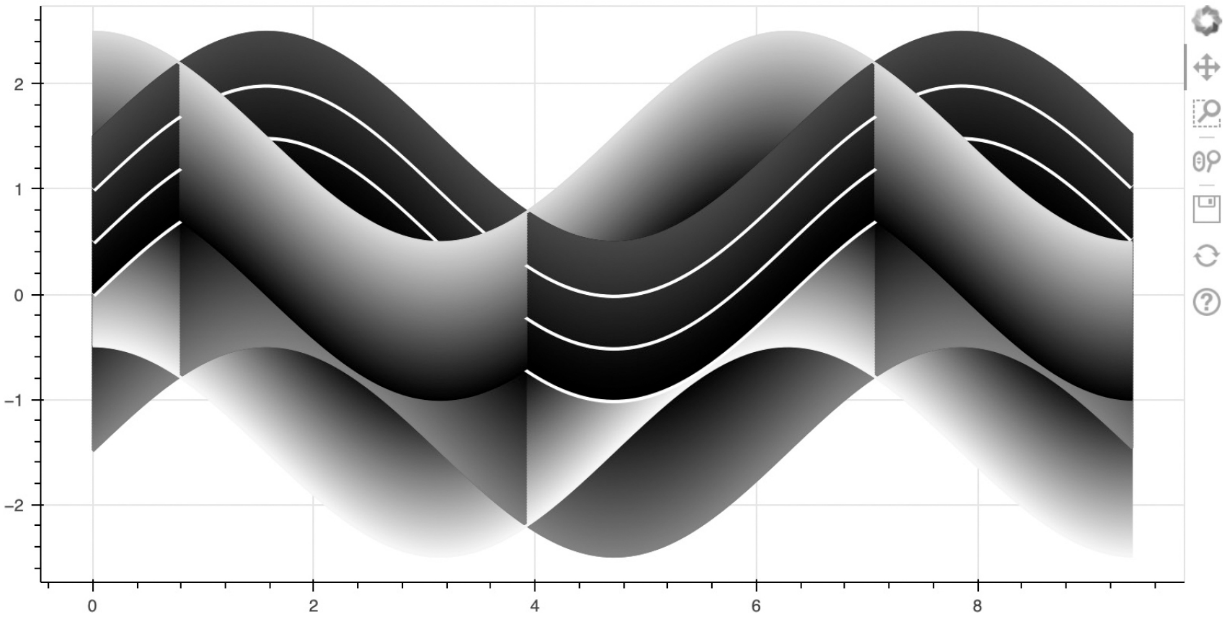


FIGURE 5.34 A Bokeh graphics sample

The next section introduces you to scikit-learn, which is a powerful Python-based library that supports many algorithms for machine learning. After you have read the short introduction, subsequent sections contain Python code samples that combine Pandas, Matplotlib, and scikit-learn built-in datasets.

INTRODUCTION TO SCIKIT-LEARN

Since this book is about data visualization, you might be wondering why this chapter contains an introduction to scikit-learn. The reason is straightforward: the easy introduction to some scikit-learn functionality is possible without a more formal learning process. In addition, this knowledge will bode well if you decide to delve into machine learning (and perhaps this section will provide additional motivation to do so).

However, a thorough understanding of scikit-learn involves significantly more time and effort, especially if you plan to learn the details of the scikit-learn machine learning algorithms. However, if you are not interested in learning about scikit-learn at this point in time, you can skip this section and perhaps return to it when you are interested in learning this material.

Scikit-learn (which is installed as `sklearn`) is Python's premier general-purpose machine learning library, and its home page is here:

https://scikit-learn.org/stable/

Before we discuss any code samples, please keep in mind that scikit-learn is an immensely useful Python library that supports a huge number of machine learning algorithms. In particular, scikit-learn supports many classification algorithms, such as logistic regression, Naive Bayes, decision trees, random forests, and SVMs (support vector machines). Although entire books are available that are dedicated to scikit-learn, this chapter contains only a few pages of scikit-learn material.

If you decide that you want to acquire a deep level of knowledge about scikit-learn, navigate to the webpages that contain detailed documentation for scikit-learn. Moreover, if you have "how to" questions involving scikit-learn, you can almost always find suitable answers on stackoverflow.

Scikit-learn is well-suited for classification tasks as well as regression and clustering tasks in machine learning. Scikit-learn supports a vast collection of machine learning algorithms, including linear regression, logistic regression, kNN (k Nearest Neighbor), kMeans, decision trees, random forests, MLPs (Multi-Layer Perceptrons), and SVMs (Support Vector Machines).

Moreover, scikit-learn supports dimensionality reduction techniques such as PCA (Principal Component Analysis), hyper parameter tuning, and methods for scaling data; it is suitable for preprocessing data, cross-validation, and so forth.

Machine learning code samples often contain a combination of scikit-learn, NumPy, Pandas, and Matplotlib. In addition, scikit-learn provides various built-in datasets that we can display visually. One of those datasets is the `Digits` dataset, which is the topic of the next section.

The next section of this chapter provides several Python code samples that contain a combination of Pandas, Matplotlib, and the scikit-learn built-in `Digits` dataset.

THE DIGITS DATASET IN SCIKIT-LEARN

The `Digits` dataset in scikit-learn comprises 1,797 small 8x8 images: each image is a hand-written digit, which is also the case for the `MNIST` dataset. Listing 5.41 displays the content of `load_digits1.py` that illustrates how to plot the `Digits` dataset.

Listing 5.41: load_digits1.py

```
from scikit-learn import datasets

# Load in the 'digits' data
digits = datasets.load_digits()

# Print the 'digits' data
print(digits)
```

Listing 5.41 is straightforward: after importing the `datasets` module, the variable `digits` is initialized with the contents of the `Digits` dataset. The `print()` statement displays the content of the `digits` variable, which is displayed here:

```
{images': array(
     [[[0.,   0.,   5., ...,       1.,    0.,   0.],
       [0.,   0.,  13., ...,  15.,   5.,    0.],
       [0.,   3.,  15., ...,  11.,   8.,    0.],
       ...,
       [0.,   4.,  11., ...,  12.,   7.,    0.],
       [0.,   2.,  14., ...,  12.,   0.,    0.],
       [0.,   0.,   6., ...,   0.,   0.,    0.]]),
'target': array([0, 1, 2, ..., 8, 9, 8]), 'frame': None,
'feature_names': ['pixel_0_0', 'pixel_0_1', 'pixel_0_2',
'pixel_0_3', 'pixel_0_4', 'pixel_0_5', 'pixel_0_6',
'pixel_0_7', 'pixel_1_0', 'pixel_1_1', 'pixel_1_2',
'pixel_1_3', 'pixel_1_4', 'pixel_1_5', 'pixel_1_6',
'pixel_1_7', 'pixel_2_0', 'pixel_2_1', 'pixel_2_2',
'pixel_2_3', 'pixel_2_4', 'pixel_2_5', 'pixel_2_6',
'pixel_2_7', 'pixel_3_0', 'pixel_3_1', 'pixel_3_2',
'pixel_3_3', 'pixel_3_4', 'pixel_3_5', 'pixel_3_6',
'pixel_3_7', 'pixel_4_0', 'pixel_4_1', 'pixel_4_2',
'pixel_4_3', 'pixel_4_4', 'pixel_4_5', 'pixel_4_6',
'pixel_4_7', 'pixel_5_0', 'pixel_5_1', 'pixel_5_2',
'pixel_5_3', 'pixel_5_4', 'pixel_5_5', 'pixel_5_6',
'pixel_5_7', 'pixel_6_0', 'pixel_6_1', 'pixel_6_2',
'pixel_6_3', 'pixel_6_4', 'pixel_6_5', 'pixel_6_6',
'pixel_6_7', 'pixel_7_0', 'pixel_7_1', 'pixel_7_2',
'pixel_7_3', 'pixel_7_4', 'pixel_7_5', 'pixel_7_6',
'pixel_7_7'], 'target_names': array([0, 1, 2, 3, 4, 5, 6,
7, 8, 9]), 'images': array([[[ 0.,  0.,  5., ...,  1.,
0.,  0.],
        [ 0.,  0., 13., ..., 15.,  5.,  0.],
        [ 0.,  3., 15., ..., 11.,  8.,  0.],
// data omitted for brevity
])}
```

Listing 5.42 displays the content of `load_digits2.py` that illustrates how to plot one digit of the `Digits` dataset (which you can change to display a different digit).

Listing 5.42: load_digits2.py

```
from scikit-learn.datasets import load_digits
from matplotlib import pyplot as plt

digits = load_digits()
#set interpolation='none'

fig = plt.figure(figsize=(3, 3))
plt.imshow(digits['images'][66], cmap="gray",
interpolation='none')
plt.show()
```

Listing 5.42 imports the `load_digits` class from scikit-learn to initialize the variable `digits` with the contents of the `Digits` dataset that is available in scikit-learn. The next portion of Listing 5.46 initializes the variable `fig` and invokes the method `imshow()` of the `plt` class to display a number in the digits dataset.

Figure 5.35 shows a plot of one of the digits in the `Digits` dataset based on the code in Listing 5.42.

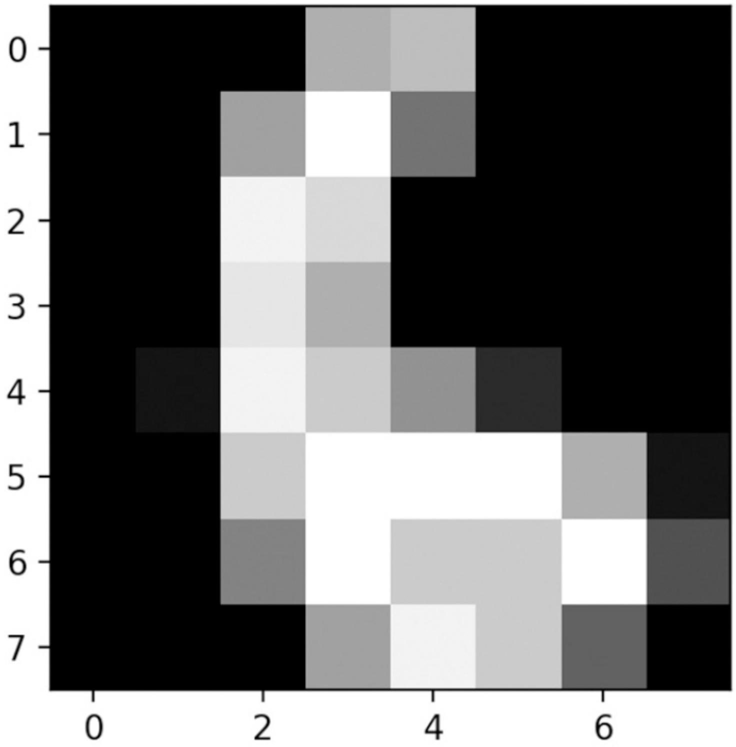


FIGURE 5.35 A digit in the scikit-learn Digits dataset

Listing 5.43 displays the content of `scikit-learn_digits.py` that illustrates how to access the `Digits` dataset in scikit-learn.

Listing 5.43: sklearn_digits.py

```
from scikit-learn import datasets

digits = datasets.load_digits()
print("digits shape:",digits.images.shape)
print("data shape:",digits.data.shape)

n_samples, n_features = digits.data.shape
print("(samples,features):", (n_samples,
n_features))

import matplotlib.pyplot as plt
```

```
#plt.imshow(digits.images[-1], cmap= plt.cm
.gray_r)
#plt.show()

plt.imshow(digits.images[0], cmap= plt.cm .binary,
interpolation='nearest')
plt.show()
```

Listing 5.43 starts with one `import` statement followed by the variable `digits` that contains the `Digits` dataset. The output from Listing 5.43 is here:

```
digits shape: (1797, 8, 8)
data    shape: (1797, 64)
(samples,features): (1797, 64)
```

Figure 5.36 shows the images in the `Digits` dataset based on the code in Listing 5.43.

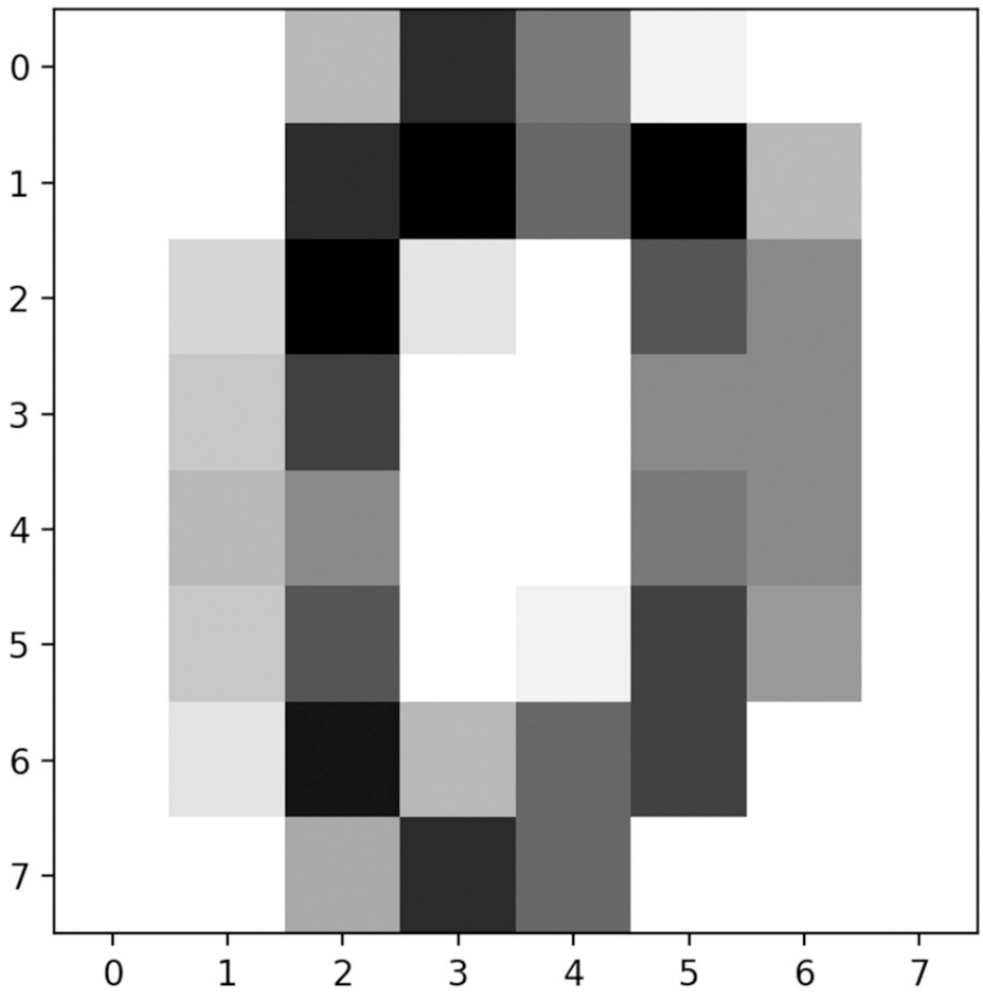


FIGURE 5.36 The digits in the Digits dataset

THE IRIS DATASET IN SCIKIT-LEARN (1)

Listing 5.44 displays the content of `sklearn_iris.py` that illustrates how to access the `Iris` dataset in scikit-learn.

In addition to support for machine learning algorithms, scikit-learn provides various built-in datasets that you can access with one line of code. In fact, Listing 5.44 displays the content of `scikit-learn_iris1.py` that illustrates how you can easily load the `Iris` dataset into a Pandas data frame.

Listing 5.44: scikit-learn_iris.py

```
import numpy as np
import pandas as pd
from scikit-learn.datasets import load_iris

iris = load_iris()

print("=> iris keys:")
for key in iris.keys():
  print(key)
print()

#print("iris dimensions:")
#print(iris.shape)
#print()

print("=> iris feature names:")
for feature in iris.feature_names:
  print(feature)
print()

X = iris.data[:, [2, 3]]
y = iris.target
print('=> Class labels:', np.unique(y))
print()

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() +
.5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() +
.5

print("=> target:")
print(iris.target)
print()

print("=> all data:")
print(iris.data)
```

Listing 5.44 contains several `import` statements and then initializes the variable `iris` with the `Iris` dataset. Next, a loop displays the keys in the dataset, followed by another loop that displays the feature names.

The next portion of Listing 5.44 initializes the variable `X` with the feature values in columns 2 and 3, and then initializes the variable `y` with the values of the target column.

The variable `x_min` is initialized as the minimum value of column 0 and then an additional 0.5 is subtracted from `x_min`. Similarly, the variable `x_max` is initialized as the maximum value of column 0 and then an additional 0.5 is added to `x_max`. The variables `y_min` and `y_max` are the counterparts to `x_min` and `x_max`, applied to column 1 instead of column 0.

Launch the code in Listing 5.44, and you will see the following output (truncated to save space):

```
Pandas df1:

=> iris keys:
data
target
target_names
DESCR
feature_names
filename
```

```
=> iris feature names:
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)

=> Class labels: [0 1 2]
=> x_min: 0.5 x_max: 7.4
=> y_min: -0.4 y_max: 3.0

=> target:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2]

=> all data:
[[5.1 3.5 1.4 0.2]
  [4.9 3. 1.4 0.2]
  [4.7 3.2 1.3 0.2]
  // details omitted for brevity
  [6.5 3. 5.2 2. ]
  [6.2 3.4 5.4 2.3]
  [5.9 3. 5.1 1.8]]
```

Scikit-Learn, Pandas, and the Iris Dataset

Listing 5.45 displays the content of `pandas_iris.py` that illustrates how to load the contents of the `Iris` dataset into a Pandas data frame.

Listing 5.45: pandas_iris.py

```
import numpy as np
import pandas as pd
from scikit-learn.datasets import load_iris

iris = load_iris()

print("=> IRIS feature names:")
for feature in iris.feature_names:
  print(feature)
print()

# Create a dataframe with the feature variables
df = pd.DataFrame(iris.data,
columns=iris.feature_names)

print("=> number of rows:")
print(len(df))
print()

print("=> number of columns:")
print(len(df.columns))
print()

print("=> number of rows and columns:")
print(df.shape)
print()

print("=> number of elements:")
print(df.size)
print()

print("=> IRIS details:")
print(df.info())
print()

print("=> top five rows:")
print(df.head())
print()

X = iris.data[:, [2, 3]]
y = iris.target
print('=> Class labels:', np.unique(y))
```

Listing 5.45 contains several `import` statements and then initializes the variable `iris` with the Iris dataset. Next, a loop displays the feature names. The next code snippet initializes the variable `df` as a Pandas data frame that contains the data from the `Iris` dataset.

The next block of code invokes some attributes and methods of a Pandas data frame to display the number of rows, columns, and elements in the data frame, as well as the details of the `Iris` dataset, the first five rows, and the unique labels in the `Iris` dataset. Launch the code in Listing 5.45, and you will see the following output:

```
=> IRIS feature names:
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)

=> number of rows:
```

```
150

=> number of columns:
4

=> number of rows and columns:
(150, 4)

=> number of elements:
600

=> IRIS details:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
sepal length (cm)    150 non-null float64
sepal width (cm)     150 non-null float64
petal length (cm)    150 non-null float64
petal width (cm)     150 non-null float64
dtypes: float64(4)
memory usage: 4.8 KB
None

=> top five rows:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

=> Class labels: [0 1 2]
```

THE IRIS DATASET IN SCIKIT-LEARN (2)

The `Iris` dataset in scikit-learn consists of the lengths of three different types of `Iris`-based petals and sepals: `Setosa`, `Versicolour`, and `Virginica`. These numeric values are stored in a 150x4 `NumPy.ndarray`.

Note that the rows in the `Iris` dataset are the sample images, and the columns consist of the values for the Sepal Length, Sepal Width, Petal Length, and Petal Width of each image. Listing 5.46 displays the content of `scikit-learn_iris.py` that illustrates how to display detailed information about the `Iris` dataset.

Listing 5.46: sklearn_iris.py

```
from scikit-learn import datasets
from scikit-learn.model_selection import
train_test_split

iris = datasets.load_iris()
data = iris.data

print("iris data shape: ",data.shape)
print("iris target shape:",iris.target.shape)
print("first 5 rows iris:")
print(data[0:5])
print("keys:",iris.keys())
print("")

n_samples, n_features = iris.data.shape
print('Number of samples: ', n_samples)
print('Number of features:', n_features)
print("")

print("sepal length/width and petal
length/width:")
print(iris.data[0])

import numpy as np
np.bincount(iris.target)

print("target names:",iris.target_names)

print("mean: %s " % data.mean(axis=0))
print("std: %s " % data.std(axis=0))

#print("mean: %s " % data.mean(axis=1))
#print("std: %s " % data.std(axis=1))

# load the data into train and test datasets:
X_train, X_test, y_train, y_test =
train_test_split(iris.data, iris.target,
random_state=0)
from scikit-learn.preprocessing import
StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)

# rescale the train datasest:
X_train_scaled = scaler.transform(X_train)
print("X_train_scaled
shape:",X_train_scaled.shape)

print("mean : %s " % X_train_scaled.mean(axis=0))
print("standard deviation : %s " %
X_train_scaled.std(axis=0))

import matplotlib.pyplot as plt

x_index = 3
colors = ['blue', 'red', 'green']
for label, color in
zip(range(len(iris.target_names)), colors):
  plt.hist(iris.data[iris.target==label, x_index],
    label=iris.target_names[label],

    color=color)

plt.xlabel(iris.feature_names[x_index])
plt.legend(loc='upper right')
plt.show()
```

Listing 5.46 starts with an `import` statement followed by the variables `iris` and `data`, where the latter contains the `Iris` dataset. The first half of Listing 5.48 consists of self-explanatory code, such as displaying the number of images and the number of features in the `Iris` dataset.

The second portion of Listing 5.46 imports the `StandardScaler` class in scikit-learn, which rescales each value in `X_train` by subtracting the mean and then dividing by the standard deviation. The final block of code in Listing 5.46 generates a histogram that displays some of the images in the `Iris` dataset. The output from Listing 5.46 is here:

```
iris data shape: (150, 4)
iris target shape: (150,)
first 5 rows iris:
```

```
[[5.1 3.5 1.4 0.2]
  [4.9 31.4 0.2]
  [4.7 3.2 1.3 0.2]
  [4.6 3.1 1.5 0.2]
  [53.6 1.4 0.2]]
keys: dict_keys(['target', 'target_names', 'data',
'feature_names', 'DESCR'])

Number of samples: 150
Number of features: 4

sepal length/width and petal length/width:
[5.1 3.5 1.4 0.2]
target names: ['setosa' 'versicolor' 'virginica']
mean: [5.84333333 3.054 3.75866667 1.19866667]
std: [0.82530129 0.43214658 1.75852918 0.76061262]
X_train_scaled shape: (112, 4)
mean : [ 1.21331516e-15 -4.41115398e-17
7.13714802e-17 2.57730345e-17]
standard deviation : [1. 1. 1. 1.]
```

Figure 5.37 shows the images in the `Iris` dataset based on the code in Listing 5.46.

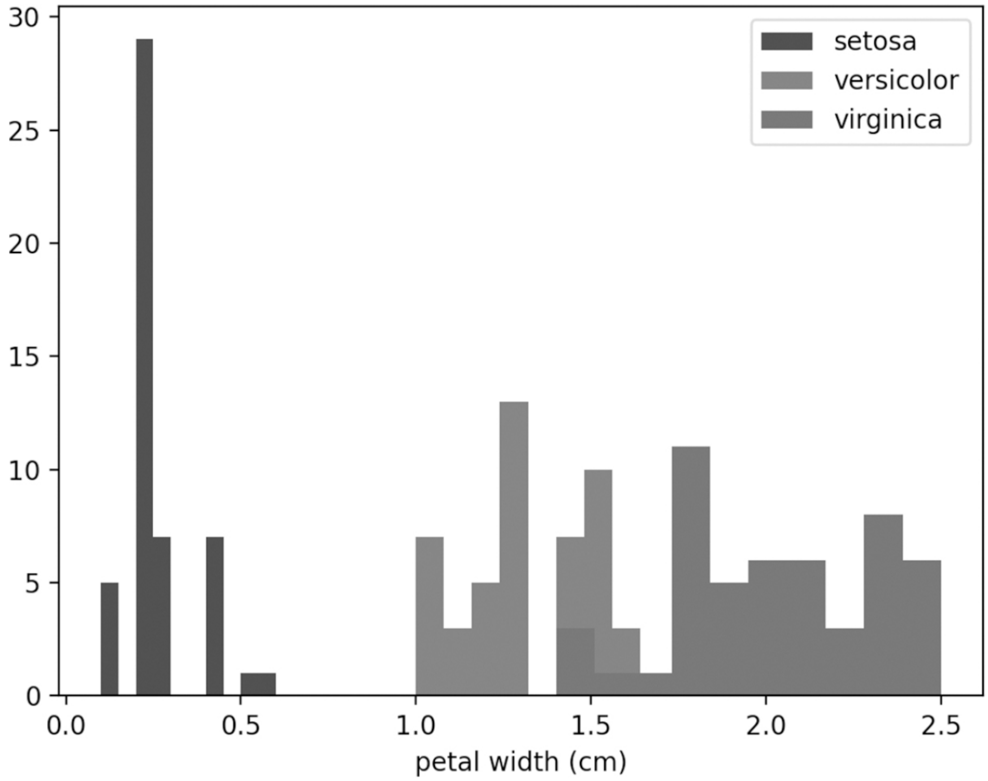


FIGURE 5.37 The Iris dataset

ADVANCED TOPICS IN SEABORN

Listing 5.47 displays the content `sns_kde_plot1.py` that shows a kde plot.

Listing 5.47: sns_kde_plot1.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

np.random.seed(1)
numerical_1 = np.random.randn(100)

np.random.seed(2)
```

```
numerical_2 = np.random.randn(100)

fig, ax = plt.subplots(figsize=(3,3))

sns.kdeplot(x=numerical_1,
           y= numerical_2,
           ax=ax,
           shade=True,
           color="blue",
           bw=1)
plt.show()
```

Listing 5.47 starts with several `import` statements and then sets an initial random seed value. This value is used for initializing the variables `numerical_1` and `numerical_2` with a set of 100 random numbers.

The next portion of Listing 5.47 initializes the variables `fig` and `ax` as subplots, followed by the Seaborn `kdeplot()` method that uses the previously initialized variables to generate a kde plot. Figure 5.38 shows the result of launching the code in Listing 5.47.

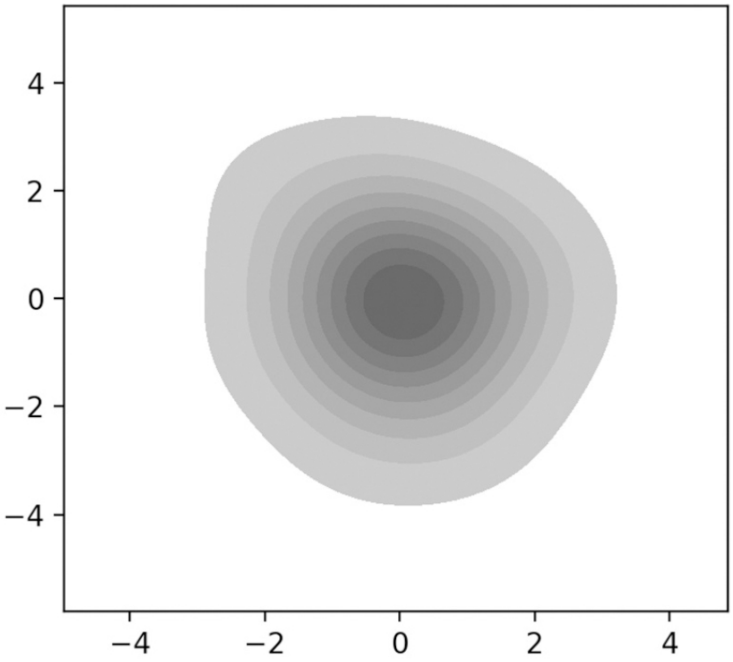


FIGURE 5.38 A Pandas data frame displayed via Seaborn

Listing 5.48 displays the content `sns_line_barchart1.py` that shows a line graph and a bar chart.

Listing 5.48: sns_line_barchart1.py

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set(style="white", rc={"lines.linewidth": 3})
fig, ax1 = plt.subplots(figsize=(5,5))
ax2 = ax1.twinx()

sns.barplot(x=['A', 'B', 'C', 'D', 'E'],
            y=[150,230,120,80,190],
            color='#224488',
            ax=ax1)

sns.lineplot(x=['X1','X2','X3','X4','X5'],
            y=[4,2,5,3,6],
            color='r',
            marker="o",
            ax=ax2)

plt.show()
sns.set()
```

Listing 5.48 starts with two import statements and then initializes some display-related parameters. Next, the variables `fig`, `ax1`, and `ax2` are initialized as subplots that will be populated with a bar plot and a line plot.

The next portion of Listing 5.48 defines the parameter values for a bar plot (i.e., 5 bar elements), such as the labels for the bar elements, their coordinates, and their color. The second plot is a line plot that performs a corresponding initialization of required parameters. Launch the code in Listing 5.48, and you will see a plot of the data in the `Titanic` dataset, as shown in Figure 5.39.

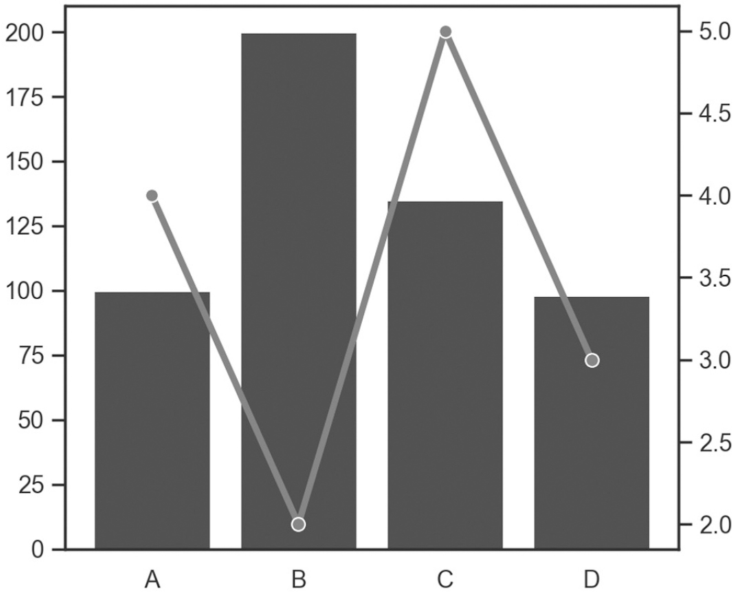


FIGURE 5.39 A line graph and bar chart

SUMMARY

This chapter started with a very short introduction to Matplotlib, along with code samples that displayed the available styles in colors in Matplotlib.

Then you learned how to render horizontal lines, slanted lines, parallel lines, and a grid of points. In addition, you learned how to load images, display checkerboard patterns, and plot trigonometric functions. Moreover, you saw how to render histograms, bar charts, pie charts, and heat maps.

Next, you saw how to create a 3D plot, how to render financial data, and render a chart with data from a sqlite3 database. In addition, you learned about Bokeh, along with an example of rendering graphics in Bokeh.

You also learned about scikit-learn, including examples of working with the Digits and Iris datasets, and also how to process images.

OceanofPDF.com

APPENDIX

WORKING WITH AWK

This appendix introduces you to the `awk` command, which is a versatile utility for manipulating data and restructuring datasets. This utility is so versatile that entire books have been written about the `awk` utility. Awk is essentially an entire programming language in a single command, which accepts standard input, gives standard output, and uses regular expressions and metacharacters in the same way other Unix commands do. This lets you plug it into other expressions and do almost anything, at the cost of adding complexity to a command string that may already be doing quite a lot already. It is almost always worthwhile to add a comment when using awk: it is so versatile that it will not be clear which of the many features you are using at a glance.

The first part of this appendix provides a very brief introduction of the `awk` command. You will learn about some built-in variables for `awk`, and also how to manipulate string variables using `awk`. Note that some of these string-related examples can also be handled using other bash commands.

The second part of this appendix shows you conditional logic, `while` loops, and `for` loops in `awk` to manipulate the rows and columns in datasets. This section also shows you how to delete lines and merge lines in datasets, and also how to print the contents of a file as a single line of text. You will see how to "join" lines and groups of lines in datasets.

The third section contains code samples that involve metacharacters and character sets in `awk` commands. You will also see how to use conditional logic in `awk` commands to determine whether to print a line of text.

The fourth section illustrates how to "split" a text string that contains multiple "." characters as a delimiter, followed by examples of `awk` to perform numeric calculations (such as addition, subtraction, multiplication, and division) in files containing numeric data. This section also shows you various numeric functions that are available in awk, as well as how to print text in a fixed set of columns.

The fifth section explains how to align columns in a dataset, and how to align and merge columns in a dataset. You will see how to delete columns, how to select a subset of columns from a dataset, and how to work with multi-line records in datasets. This section contains some one-line `awk` commands that can be useful for manipulating the contents of datasets.

The final section of this appendix has a pair of use cases involving nested quotes and date formats in structured datasets.

THE AWK COMMAND

The `awk` (Aho, Weinberger, and Kernighan) command has a C-like syntax and you can use this utility to perform very complex operations on numbers and text strings.

As a side comment, there is also the `gawk` command that is GNU `awk`, as well as the `nawk` command is "new" `awk` (neither command is discussed in this book). One advantage of `nawk` is that it allows you to externally set the value of an internal variable.

Built-In Variables That Control awk

The `awk` command provides variables that you can change from their default values to control how `awk` performs operations. Examples of such variables (and their default values) include `FS (" ")`, `RS ("\n")`, `OFS (" ")`, `ORS ("\n")` , `SUBSEP`, and `IGNORECASE`. The variables `FS` and `RS` specify the field separator and record separator, whereas the variables `OFS` and `ORS` specify the output field separator and the output record separator, respectively.

You can think of the field separators as delimiters/IFS we used in other commands earlier. The record separators behave in a way similar to how sed treats individual lines; for example, sed can match or delete a range of lines instead of matching or deleting something that matches a regular expression (and the default awk record separator is the newline character, so by default awk and sed have similar ability to manipulate and reference lines in a text file).

As a simple example, you can print a blank line after each line of a file by changing the `ORS`, from default of one newline to two newlines, as shown here:

```
cat columns.txt | awk 'BEGIN { ORS ="\n\n" } ; {
print $0 }'
```

Other built-in variables include `FILENAME` (the name of the file that `awk` is currently reading), `FNR` (the current record number in the current file), `NF` (the number of fields in the current input record), and `NR` (the number of input records `awk` has processed since the beginning of the program's execution).

Consult the online documentation for additional information regarding these (and other) arguments for the `awk` command.

How Does the awk Command Work?

The `awk` command reads the input files one record at a time (by default, one record is one line). If a record matches a pattern, then an action is performed (otherwise, no action is performed). If the search pattern is not given, then `awk` performs the given actions for each record of the input. The default behavior if no action is given is to print all the records that match the given pattern. Finally, empty braces without any action does nothing; i.e., it will not perform the default printing operation. Note that each statement in actions should be delimited by semicolon.

To make the preceding paragraph clearer, here are some simple examples involving text strings and the `awk` command (the results are displayed after each code snippet). The `-F` switch sets the field separator to whatever follows it, in this case, a space. Switches will often provide a shortcut to an action that normally needs a command inside a '`BEGIN{}` block):

```
x="a b c d e"
echo $x |awk -F" " '{print $1}'
a
echo $x |awk -F" " '{print NF}'
5
echo $x |awk -F" " '{print $0}'
a b c d e
```

```
echo $x |awk -F" " '{print $3, $1}'
c a
```

Now let’s change the `FS` (record separator) to an empty string to calculate the length of a string, this time using the `BEGIN{}` syntax:

```
echo "abc" | awk 'BEGIN { FS = "" } ; { print NF
}'
3
```

The following example illustrates several equivalent ways to specify `test.txt` as the input file for an `awk` command:

```
awk < test.txt '{ print $1 }'
awk '{ print $1 }' < test.txt
awk '{ print $1 }' test.txt
```

Yet another way is shown here (but as we have discussed earlier, it can be inefficient, so only do it if the cat is adding value in some way):

```
cat test.txt | awk '{ print $1 }'
```

This simple example of four ways to do the same task should illustrate why commenting awk calls of any complexity is almost always a good idea. The next person to look at your code may not know/remember the syntax you are using.

ALIGNING TEXT WITH THE PRINTF() STATEMENT

Since awk is a programming language inside a single command, it also has its own way of producing formatted output via the printf() statement.

Listing A.1 displays the contents of `columns2.txt`, and Listing A.2 displays the content of the shell script `AlignColumns1.sh` that shows you how to align the columns in a text file.

Listing A.1: columns2.txt

```
one two
three four
one two three four
five six
one two three
four five
```

Listing A.2: AlignColumns1.sh

```
awk '
{
  # left-align $1 on a 10-char column
  # right-align $2 on a 10-char column
  # right-align $3 on a 10-char column
  # right-align $4 on a 10-char column
  printf("%-10s*%10s*%10s*%10s*\n", $1, $2, $3, $4)
}
' columns2.txt
```

Listing A.2 contains a `printf()` statement that displays the first four fields of each row in the file `columns2.txt`, where each field is 10 characters wide.

The output from launching the code in Listing A.2 is here:

```
one       *       two*          *          *
three     *      four*          *          *
one       *       two*     three*      four*
five      *       six*          *          *
one       *       two*     three*          *
four      *      five*          *          *
```

The `printf()` statement is reasonably powerful and as such has its own syntax, which is beyond the scope of this appendix. A search online can find the manual pages and also discussions of “how to do X with `printf()`.”

CONDITIONAL LOGIC AND CONTROL STATEMENTS

Like other programming languages, `awk` provides support for conditional logic (if/else) and control statements (for/while loops). awk is the only way to put conditional logic inside a piped command stream without creating, installing and adding to the path a custom executable shell script. The following code block shows you how to use if/else logic:

```
echo "" | awk '
BEGIN { x = 10 }
{
  if (x % 2 == 0) }
      print "x is even"
  }
  else }
      print "x is odd"
  }
```

```
}
'
```

The preceding code block initializes the variable `x` with the value 10 and prints “`x is even`” if `x` is divisible by 2; otherwise, it prints “`x is odd`.”

The while Statement

The following code block illustrates how to use a `while` loop in `awk`:

```
echo "" | awk '
{
  x = 0
  while(x < 4) {
    print "x:",x
    x = x + 1
  }
}
'
```

The preceding code block generates the following output:

```
x:0
x:1
x:2
x:3
```

The following code block illustrates how to use a `do while` loop in `awk`:

```
echo "" | awk '
{
  x = 0
  do {
    print "x:",x
    x = x + 1
  } while(x < 4)
}
'
```

The preceding code block generates the following output:

```
x:0
x:1
x:2
x:3
```

A for Loop in Awk

Listing A.3 displays the content of `Loop.sh` that illustrates how to print a list of numbers in a loop. Note that “i++” is another way of writing “I=I+1” in awk (and most C-derived languages).

Listing A.3: Loop.sh

```
echo "" | awk '
BEGIN {}
{
  for(i=0; i<5; i++) {
    printf("%3d", i)
  }
}
END { print "\n" }
'
```

Listing A.3 contains a `for` loop that prints numbers on the same line via the `printf()` statement. Notice that a new line is printed only in the `END` block of the code. The output from Listing A.3 is here:

```
0 1 2 3 4
```

A for Loop with a break Statement

The following code block illustrates how to use a `break` statement in a `for` loop in `awk`:

```
echo "" | awk '
{
  for(x=1; x<4; x++) {
    print "x:",x
    if(x == 2) {
       break;
    }
  }
}
'
```

The preceding code block prints output only until the variable x has the value 2, after which the loop exits (because of the break inside the conditional logic). The following output is displayed:

```
x:1
```

The next and continue Statements

The following code snippet illustrates how to use `next` and `continue` in a `for` loop in `awk`:

```
awk '
```

```
{
  /expression1/ { var1 = 5; next }
  /expression2/ { var2 = 7; next }
  /expression3/ { continue }
  // some other code block here
' somefile
```

When the current line matches `expression1`, then `var1` is assigned the value 5 and `awk` reads the next input line: hence, `expression2` and `expression3` will not be tested. If `expression1` does not match and `expression2` *does* match, then `var2` is assigned the value 7 and then `awk` will read the next input line. If only `expression3` results in a positive match, then `awk` skips the remaining block of code and processes the next input line.

DELETING ALTERNATE LINES IN DATASETS

Listing A.4 displays the contents of `linepairs.csv`, and Listing A.5 displays the content of `deletelines.sh` that illustrates how to print alternating lines from the dataset `linepairs.csv` that have exactly two columns.

Listing A.4: linepairs.csv

```
a,b,c,d
e,f,g,h
1,2,3,4
5,6,7,8
```

Listing A.5: deletelines.sh

```
inputfile="linepairs.csv"
outputfile="linepairsdeleted.csv"
awk ' NR%2 {printf "%s", $0; print ""; next}' <
$inputfile > $outputfile
```

Listing A.5 checks if the current record number `NR` is divisible by 2, in which case it prints the current line and skips the next line in the dataset. The output is redirected to the specified output file, the contents of which are here:

```
a,b,c,d
1,2,3,4
```

A slightly more common task involves merging consecutive lines, which is the topic of the next section.

MERGING LINES IN DATASETS

Listing A.6 displays the contents of `columns.txt`, and Listing A.7 displays the content of `ColumnCount1.sh` that illustrates how to print the lines from the text file `columns.txt` that have exactly two columns.

Listing A.6: columns.txt

```
one two three
one two
one two three four
one
one three
one four
```

Listing A.7: ColumnCount1.sh

```
awk '
{
  if( NF == 2 ) { print $0 }
}
' columns.txt
```

Listing A.7 is straightforward: if the current record number is even, then the current line is printed (i.e., odd-numbered rows are skipped). The output from launching the code in Listing A.7 is here:

```
one two
one three
one four
```

If you want to display the lines that do *not* contain 2 columns, use the following code snippet:

```
if( NF != 2 ) { print $0 }
```

Printing File Contents as a Single Line

The contents of `test4.txt` are here (note the blank lines):

```
abc

def

abc

abc
```

The following code snippet illustrates how to print the contents of `test4.txt` as a single line:

```
awk '{printf("%s", $0)}' test4.txt
```

The output of the preceding code snippet is here. See if you can tell what is happening before reading the explanation in the next paragraph:

```
Abcdefabcabc
```

Explanation: `%s` here is the record separator syntax for `printf()`, with the end quotation mark after it means the record separator is the empty field “”. Our default record separator for `awk` is `/n` (newline), so the `printf()` statement strips out all the new lines. The blank rows will vanish entirely, as all they have is the new line, so the result is that any actual text will be merged together with nothing between them.

Had we added a space between the `%s` and the ending quotation mark, there would be a space between each character block, plus an extra space for each new line. Notice how the following comment improves the comprehension of the code snippet:

```
# Merging all text into a single line by removing
the newlines
awk '{printf("%s", $0)}' test4.txt
```

Joining Groups of Lines in a Text File

Listing A.8 displays the contents of `digits.txt`, and Listing A.9 displays the content of `digits.sh` that “joins” three consecutive lines of text in the file `digits.txt`.

Listing A.8: digits.txt

```
1
2
```

```
3
4
5
6
7
8
9
```

Listing A.9: digits.sh

```
awk -F" " '{
  printf("%d",$0)
  if(NR % 3 == 0) { printf("\n") }
}' digits.txt
```

Listing A.9 prints three consecutive lines of text on the same line, after which a linefeed is printed. This has the effect of "joining" every three consecutive lines of text. The output from launching `digits.sh` is here:

```
123
456
789
```

Joining Alternate Lines in a Text File

Listing A.10 displays the contents of `columns2.txt`, and Listing A.11 displays the content of `JoinLines.sh` that "joins" two consecutive lines of text in the file `columns2.txt`.

Listing A.10: columns2.txt

```
one two
```

```
three four
one two three four
five six
one two three
four five
```

Listing A.11: JoinLines.sh

```
awk '
{
   printf("%s",$0)
   if( $1 !~ /one/) { print " " }
}
' columns2.txt
```

The output from launching Listing A.11 is here:

```
one two three four
one two three four five six
one two three four five
```

Notice that the code in Listing A.11 depends on the presence of the string “one” as the first field in alternating lines of text; we are merging based on matching a simple pattern, instead of tying it to record combinations.

To merge each pair of lines instead of merging based on matching a pattern, use the modified code in Listing A.12.

Listing A.12: JoinLines2.sh

```
awk '
BEGIN { count = 0 }
{
   printf("%s",$0)
```

```
    if( ++count % 2 == 0) { print " " }
} columns2.txt
```

Yet another way to “join” consecutive lines is shown in Listing A.13, where the input file and output file refer to files that you can populate with data. This is another example of an `awk` command that might be a puzzle if encountered in a program without a comment. It is doing exactly the same thing as Listing A.12, but its purpose is less obvious because of the more compact syntax.

Listing A.13: JoinLines2.sh

```
inputfile="linepairs.csv"
outputfile="linepairsjoined.csv"
awk ' NR%2 {printf "%s,", $0; next;}1' <
$inputfile > $outputfile
```

MATCHING WITH METACHARACTERS AND CHARACTER SETS

If we can match a simple pattern, you can also match a regular expression. Listing A.14 displays the content of `Patterns1.sh` that uses metacharacters to match the beginning and the end of a line of text in the file `columns2.txt`.

Listing A.14: Patterns1.sh

```
awk '
    /^f/ { print $1 }
    /two $/ { print $1 }
' columns2.txt
```

The output from launching Listing A.14 is here:

```
one
five
four
```

Listing A.15 displays the content of `RemoveColumns.txt` with lines that contain a different number of columns.

Listing A.15: columns3.txt

```
123 one two
456 three four
one two three four
five 123 six
one two three
four five
```

Listing A.16 displays the content of `MatchAlpha1.sh` that matches text lines that start with alphabetic characters, as well as lines that contain numeric strings in the second column.

Listing A.16: MatchAlpha1.sh

```
awk '
{
   if( $0 ~ /^[0-9]/) { print $0 }
   if( $0 ~ /^[a-z]+ [0-9]/) { print $0 }
}
' columns3.txt
```

The output from Listing A.16 is here:

```
123 one two
```

```
456 three four
five 123 six
```

PRINTING LINES USING CONDITIONAL LOGIC

Listing A.17 displays the content of `products.txt` that contains three columns of information.

Listing A.17: products.txt

```
MobilePhone   400    new
Tablet        300    new
Tablet        300    used
MobilePhone   200    used
MobilePhone   100    used
```

The following code snippet prints the lines of text in `products.txt` whose second column is greater than `300`:

```
awk '$2 > 300' products.txt
```

The output of the preceding code snippet is here:

```
MobilePhone   400    new
```

The following code snippet prints the lines of text in `products.txt` whose product is “new:”

```
awk '($3 == "new")' products.txt
```

The output of the preceding code snippet is here:

```
MobilePhone   400    new
```

```
Tablet        300    new
```

The following code snippet prints the first and third columns of the lines of text in `products.txt` whose cost equals `300`:

```
awk ' $2 == 300 { print $1, $3 }' products.txt
```

The output of the preceding code snippet is here:

```
Tablet new
Tablet used
```

The following code snippet prints the first and third columns of the lines of text in `products.txt` that start with the string `Tablet`:

```
awk '/^Tablet/ { print $1, $3 }' products.txt
```

The output of the preceding code snippet is here:

```
Tablet new
Tablet used
```

SPLITTING FILE NAMES WITH AWK

Listing A.18 displays the content of `SplitFilename2.sh` that illustrates how to split a filename containing the "." character to increment the numeric value of one of the components of the file name. Note that this code only works for a file name with exactly the expected syntax.

Listing A.18: SplitFilename2.sh

```
echo "05.20.144 q.az .1.zip" | awk -F"." '
{
  f5=$5 + 1
  printf("%s.%s.%s.%s.%s.%s",$1,$2,$3,$4,f5,$6)
}'
```

The output from Listing A.18 is here:

```
05.20.144q.az.2.zip
```

WORKING WITH POSTFIX ARITHMETIC OPERATORS

Listing A.19 displays the content of `mixednumbers.txt` that contains postfix operators, which means numbers where the negative (or positive) sign appears at the end of a column value instead of the beginning of the number.

Listing A.19: mixednumbers.txt

```
324.000-|10|983.000-
453.000-|30|298.000-
783.000-|20|347.000-
```

Listing A.20 displays the content of `AddSubtract1.sh` that illustrates how to add the rows of numbers in Listing A.19.

Listing A.20: AddSubtract1.sh

```
myFile="mixednumbers.txt"

awk -F"|" '
BEGIN { line = 0; total = 0 }
{
  split($1, arr, "-")
  f1 = arr[1]
  if($1 ~ /-/) { f1 = -f1 }
  line += f1

  split($2, arr, "-")
  f2 = arr[1]
  if($2 ~ /-/) { f2 = -f2 }
  line += f2

  split($3, arr, "-")
  f3 = arr[1]
  if($3 ~ /-/) { f3 = -f3 }
  line += f3

  printf("f1: %d f2: %d f3: %d line:
%d\n",f1,f2,f3, line)
  total += line
  line = 0
}
END { print "Total: ",total }
' $myfile
```

The output from Listing A.20 is here. See if you can work out what the code is doing before reading the explanation that follows:

```
f1: -324 f2: 10 f3: -983 line: -1297
f1: -453 f2: 30 f3: -298 line: -721
f1: -783 f2: 20 f3: -347 line: -1110
Total: -3128
```

The code assumes we know the format of the file. The `split()` function turns each field record into a length two vector: the first position is a number and second position either an empty value or a dash, and then captures the first position number into a variable. The `if` statement just sees if the original field has a dash in it. If the field has a hyphen ("-"), then the numeric variable is made negative; otherwise, it is left alone. Then it adds up the values in the line.

NUMERIC FUNCTIONS IN AWK

The `int(x)` function returns the integer portion of a number. If the number is not already an integer, it falls between two integers. Of the two possible integers, the function will return the one closest to zero. This is different from a rounding function, which chooses the closer integer.

For example, `int(3)` is 3, `int(3.9)` is 3, int(-3.9) is -3, and `int(-3)` is -3 as well. An example of the `int(x)` function in an `awk` command is here:

```
awk 'BEGIN {
  print int(3.534);
  print int(4);
  print int(-5.223);
  print int(-5);
}'
```

The output is here:

```
3
4
-5
-5
```

The `exp(x)` function gives you the exponential of `x`, or reports an error if `x` is out of range. The range of values `x` can have depends on your machine's floating point representation.

```
awk 'BEGIN{
  print exp(123434346);
  print exp(0);
  print exp(-12);
}'
```

The output is here:

```
inf
1
6.14421e-06
```

The `log(x)` function gives you the natural logarithm of `x`, if `x` is positive; otherwise, it reports an error (`inf` means infinity and `nan` in output means “not a number”).

```
awk 'BEGIN{
  print log(12);
  print log(0);
  print log(1);
  print log(-1);
}'
```

The output is here:

```
2.48491
-inf
0
nan
```

The `sin(x)` function gives you the sine of `x` and `cos(x)` gives you the cosine of `x`, with `x` in radians:

```
awk 'BEGIN {
  print cos(90);
  print cos(45);
}'
```

The output is here:

```
-0.448074
0.525322
```

The `rand()` function gives you a random number. The values of `rand()` are uniformly-distributed between 0 and 1: the value is never 0 and never 1. Often, you want random integers instead. Here is a user-defined function you can use to obtain a random nonnegative integer less than `n`:

```
function randint(n) {
      return int(n * rand())
}
```

The product generates a random real number greater than 0 and less than `n`. We then make it an integer (using `int`) between 0 and n - 1.

Here is an example where a similar function is used to produce random integers between 1 and n:

```
awk '
# Function to roll a simulated die.
function roll(n) { return 1 + int(rand() * n) }
# Roll 3 six-sided dice and print total number of
points.
{
      printf("%d points\n",
      roll(6)+roll(6)+roll(6))
}'
```

Note that `rand()` starts generating numbers from the same point (or "seed") each time `awk` is invoked. Hence, a program will produce the same results each time it is launched. If you want a program to do different things each time it is used, you must change the seed to a value that will be different in each run.

Use the `srand(x)` function to set the starting point, or seed, for generating random numbers to the value `x`. Each seed value leads to a particular sequence of "random" numbers. Thus, if you set the seed to the same value a second time, you will get the same sequence of "random" numbers again. If you omit the argument `x`, as in `srand()`, then the current date and time of day are used for a seed. This is how to obtain random numbers that are truly unpredictable. The return value of `srand()` is the previous seed. This makes it easy to keep track of the seeds for use in consistently reproducing sequences of random numbers.

The `time()` function (not in all versions of `awk`) returns the current time in seconds since January 1, 1970. The function `ctime()` (not in all versions of `awk`) takes a numeric argument in seconds and returns a string representing the corresponding date, suitable for printing or further processing.

The `sqrt(x)` function gives you the positive square root of `x`. It reports an error if x is negative. Thus, `sqrt(4)` is 2.

```
awk 'BEGIN{
    print sqrt(16);
    print sqrt(0);
    print sqrt(-12);
}'
```

The output is here:

```
4
0
Nan
```

ONE-LINE AWK COMMANDS

The code snippets in this section reference the text file `short1.txt`, which you can populate with any data.

The following code snippet prints each line preceded by the number of fields in each line:

```
awk '{print NF ":" $0}' short1.txt
```

Print the right-most field in each line:

```
awk '{print $NF}' short1.txt
```

Print the lines that contain more than two fields:

```
awk '{if(NF > 2) print }' short1.txt
```

Print the value of the right-most field if the current line contains more than two fields:

```
awk '{if(NF > 2) print $NF }' short1.txt
```

Remove leading and trailing whitespaces:

```
echo " a b c " | awk '{gsub(/^[ \t]+|[
\t]+$/,"");print}'
```

Print the first and third fields in reverse order for the lines that contain at least three fields:

```
awk '{if(NF > 2) print $3, $1}' short1.txt
```

Print the lines that contain the string “one:”

```
awk '{if(/one/) print }' *txt
```

As you can see from the preceding code snippets, it is easy to extract information or subsets of rows and columns from text files using simple conditional logic and built-in variables in the `awk` command.

USEFUL SHORT AWK SCRIPTS

This section contains a set of short `awk` -based scripts for performing various operations. Some of these scripts can also be used in other shell scripts to perform more complex operations. Listing A.21 displays the content of the file `data.txt` that is used in various code samples in this section.

Listing A.21: data.txt

```
this is line one that contains more than 40
characters
this is line two
this is line three that also contains more than 40
characters
four
this is line six and the preceding line is empty

line eight and the preceding line is also empty
```

The following code snippet prints every line that is longer than 40 characters:

```
awk 'length($0) > 40' data.txt
```

Now print the length of the longest line in `data.txt`:

```
awk '{ if (x < length()) x = length() }
```

```
END { print "maximum line length is " x }' <
data.txt
```

The input is processed by the expand utility to change tabs into spaces, so the widths compared are actually the right-margin columns.

Print every line that has at least one field:

```
awk 'NF > 0' data.txt
```

The preceding code snippet illustrates an easy way to delete blank lines from a file (or rather, to create a new file similar to the old file but from which the blank lines have been removed).

Print seven random numbers from 0 to 100, inclusive:

```
awk 'BEGIN { for (i = 1; i <= 7; i++)
print int(101 * rand()) }'
```

Count the lines in a file:

```
awk 'END { print NR }' < data.txt
```

Print the even-numbered lines in the data file:

```
awk 'NR % 2 == 0' data.txt
```

If you use the expression `'NR % 2 == 1'` in the previous code snippet, the program would print the odd-numbered lines.

Insert a duplicate of every line in a text file:

```
awk '{print $0, '\n', $0}' < data.txt
```

Insert a duplicate of every line in a text file and also remove blank lines:

```
awk '{print $0, "\n", $0}' < data.txt | awk 'NF >
0'
```

Insert a blank line after every line in a text file:

```
awk '{print $0, "\n"}' < data.txt
```

PRINTING THE WORDS IN A TEXT STRING IN AWK

Listing A.22 displays the content of `Fields2.sh` that illustrates how to print the words in a text string using the `awk` command.

Listing A.22: Fields2.sh

```
echo "a b c d e"| awk '
{
  for(i=1; i<=NF; i++) {
     print "Field ",i,":",$i
  }
}
'
```

The output from Listing A.22 is here:

```
Field 1 : a
Field 2 : b
Field 3 : c
Field 4 : d
Field 5 : e
```

COUNT OCCURRENCES OF A STRING IN SPECIFIC ROWS

Listing A.23 and Listing A.24 display the contents `data1.csv` and `data2.csv`, respectively, and Listing A.25 displays the content of `checkrows.sh` that illustrates how to count the number of occurrences of the string “past” in column 3 in rows 2, 5, and 7.

Listing A.23: data1.csv

```
in,the,past,or,the,present
for,the,past,or,the,present
in,the,past,or,the,present
for,the,paste,or,the,future
in,the,past,or,the,present
completely,unrelated,line1
in,the,past,or,the,present
completely,unrelated,line2
```

Listing A.24: data2.csv

```
in,the,past,or,the,present
completely,unrelated,line1
for,the,past,or,the,present
completely,unrelated,line2
for,the,paste,or,the,future
in,the,past,or,the,present
in,the,past,or,the,present
completely,unrelated,line3
```

Listing A.25: checkrows.sh

```
files="`ls data*.csv| tr '\n' ' '`"
echo "List of files: $files"
awk -F"," '
```

```
( FNR==2 || FNR==5 || FNR==7 ) {
   if ( $3 ~ "past" ) { count++ }
}
END {
   printf "past: matched %d times (INEXACT) ",
   count
   printf "in field 3 in lines 2/5/7\n"
}' data*.csv
```

Listing A.25 looks for occurrences in the string `past` in columns 2, 5, and 7 because of the following code snippet:

```
( FNR==2 || FNR==5 || FNR==7 ) {
   if ( $3 ~ "past" ) { count++ }
}
```

If a match occurs, then the value of count is incremented. The `END` block reports the number of times that the string past was found in columns 2, 5, and 7. Note that strings such as `paste` and `pasted` will match the string past. The output from Listing A.25 is here:

```
List of files: data1.csv data2.csv
past: matched 5 times (INEXACT) in field 3 in
lines 2/5/7
```

The shell script `checkrows2.sh` replaces the term `$3 ~ "past"` with the term `$3 == "past"` in `checkrows.sh` to check for exact matches, which produces the following output:

```
List of files: data1.csv data2.csv
past: matched 4 times (EXACT) in field 3 in lines
2/5/7
```

PRINTING A STRING IN A FIXED NUMBER OF COLUMNS

Listing A.26 displays the content of `FixedFieldCount1.sh` that illustrates how to print the words in a text string using the `awk` command.

Listing A.26: FixedFieldCount1.sh

```
echo "aa bb cc dd ee ff gg hh"| awk '
BEGIN { colCount = 3 }
{

  for(i=1; i<=NF; i++) {
     printf("%s ", $i)
     if(i % colCount == 0) {
        print " "
     }
  }
}
'
```

The output from Listing A.26 is here:

```
aa bb cc
dd ee ff
gg hh
```

PRINTING A DATASET IN A FIXED NUMBER OF COLUMNS

Listing A.27 displays the content of `VariableColumns.txt` with lines of text that contain a different number of columns.

Listing A.27: VariableColumns.txt

```
this is line one
this is line number one
this is the third and final line
```

Listing A.28 displays the content of `Fields3.sh` that illustrates how to print the words in a text string using the `awk` command.

Listing A.28: Fields3.sh

```
awk '{printf("%s ", $0)}' | awk '
BEGIN { columnCount = 3 }
{
  for(i=1; i<=NF; i++) {
     printf("%s ", $i)
     if( i % columnCount == 0 )
       print " "
  }
}
' VariableColumns.txt
```

The output from Listing A.28 is here:

```
this is line
one this is
line number one
this is the
third and final
line
```

ALIGNING COLUMNS IN DATASETS

If you have read the preceding two examples, the code sample in this section is easy to understand: you will see how to realign columns of data that are correct in terms of their content, but have been placed in different rows (and therefore are misaligned). Listing A.29 displays the contents of `mixed-data.csv` with misaligned data values. In addition, the first line and final line in Listing A.28 are empty lines, which will be removed by the shell script in this section.

Listing A.29: mixed-data.csv

```
Sara, Jones, 1000, CA, Sally, Smith, 2000, IL,
Dave, Jones, 3000, FL, John, Jones,
4000, CA,
Dave, Jones, 5000, NY, Mike,
Jones, 6000, NY, Tony, Jones, 7000, WA
```

Listing A.30 displays the contents of `mixed-data.sh` that illustrates how to realign the dataset in Listing A.29.

Listing A.30: mixed-data.sh

```
#------------------------------------------
# 1) remove blank lines
# 2) remove line feeds
# 3) print a LF after every fourth field
# 4) remove trailing ',' from each row
#------------------------------------------

inputfile="mixed-data.csv"

grep -v "^$" $inputfile |awk -F","
'{printf("%s",$0)}' | awk '
BEGIN { columnCount = 4 }
```

```
{
  for(i=1; i<=NF; i++) {
    printf("%s ", $i)
    if( i % columnCount == 0) { print "" }
  }
}' > temp-columns

# 4) remove trailing ',' from output:
cat temp-columns | sed 's/, $//' | sed 's/ $//' >
$outputfile
```

Listing A.30 starts with a `grep` command (online tutorials about `grep` are available) that removes blank lines, followed by an `awk` command that prints the rows of the dataset as a single line of text. The second `awk` command initializes the `columnCount` variable with the value 4 in the `BEGIN` block, followed by a loop that iterates through the input fields. After four fields are printed on the same output line, a linefeed is printed, which has the effect of realigning the input dataset as an output dataset consisting of rows that have four fields. The output from Listing A.30 is here:

```
Sara, Jones, 1000, CA
Sally, Smith, 2000, IL
Dave, Jones, 3000, FL
John, Jones, 4000, CA
Dave, Jones, 5000, NY
Mike, Jones, 6000, NY
Tony, Jones, 7000, WA
```

ALIGNING COLUMNS AND MULTIPLE ROWS IN DATASETS

The preceding section showed you how to realign a dataset so that each row contains the same number of columns and also represents a single data record. The code sample in this section illustrates how to realign columns of data that are correct in terms of their content, and also place two records in each line of the new dataset. Listing A.31 displays the contents of `mixed-data2.csv` with misaligned data values, followed by Listing A.32 that displays the contents of `aligned-data2.csv` with the correctly formatted dataset.

Listing A.31: mixed-data2.csv

```
Sara, Jones, 1000, CA, Sally, Smith, 2000, IL,

Dave, Jones, 3000, FL, John, Jones,
4000, CA,
Dave, Jones, 5000, NY, Mike,
Jones, 6000, NY, Tony, Jones, 7000, WA
```

Listing A.32: aligned-data2.csv

```
Sara, Jones, 1000, CA, Sally, Smith, 2000, IL
Dave, Jones, 3000, FL, John, Jones, 4000, CA
Dave, Jones, 5000, NY, Mike, Jones, 6000, NY
Tony, Jones, 7000, WA
```

Listing A.33 displays the contents of `mixed-data2.sh` that illustrates how to realign the dataset in Listing A.31.

Listing A.33: mixed-data2.sh

```
#-----------------------------------------
# 1) remove blank lines
# 2) remove line feeds
# 3) print a LF after every 8 fields
# 4) remove trailing ',' from each row
#-----------------------------------------
inputfile="mixed-data2.txt"
outputfile="aligned-data2.txt"

grep -v "^$" $inputfile |awk -F","
'{printf("%s",$0)}' | awk '
BEGIN { columnCount = 4; rowCount = 2; currRow = 0
}
{
  for(i=1; i<=NF; i++) {
    printf("%s ", $i)
    if( i % columnCount == 0) { ++currRow }
    if(currRow > 0 && currRow % rowCount == 0)
{currRow = 0; print ""}
  }
}' > temp-columns

# 4) remove trailing ',' from output:
cat temp-columns | sed 's/, $//' | sed 's/ $//' >
$outputfile
```

Listing A.33 is very similar to Listing A.30. The key idea is to print a linefeed character after a pair of “normal” records have been processed, which is implemented via the code that is shown in bold in Listing A.33.

Now you can generalize Listing A.33 very easily by changing the initial value of the `rowCount` variable to any other positive integer, and the code will work correctly without any further modification. For example, if you initialize `rowCount` to the value `5`, then every row in the new dataset (with the possible exception of the final output row) will contain five “normal” data records.

REMOVING A COLUMN FROM A TEXT FILE

Listing A.34 displays the contents of `VariableColumns.txt` with lines of text that contain a different number of columns.

Listing A.34: VariableColumns.txt

```
this is line one
this is line number one
this is the third and final line
```

Listing A.35 displays the contents of `RemoveColumn.sh` that removes the first column from a text file.

Listing A.35: RemoveColumn.sh

```
awk '{ for (i=2; i<=NF; i++) printf "%s ", $i;
printf "\n"; }' products.txt
```

The loop is between 2 and `NF`, which iterates over all the fields except for the first field. In addition, `printf()` explicitly adds new lines. The output of the preceding code snippet is here:

```
400 new
300 new
300 used
200 used
```

```
100 used
```

SUBSETS OF COLUMN-ALIGNED ROWS IN DATASETS

Listing A.35 showed you how to align the rows of a dataset, and the code sample in this section illustrates how to extract a subset of the existing columns and a subset of the rows. Listing A.36 displays the contents of `sub-rows-cols.txt` of the desired dataset, which contains two columns from every even row of the file `aligned-data.txt`.

Listing A.36: sub-rows-cols.txt

```
Sara, 1000
Dave, 3000
Dave, 5000
Tony, 7000
```

Listing A.37 displays the contents of `sub-rows-cols.sh` that illustrates how to generate the dataset in Listing A.36. Most of the code is the same as Listing A.33, with the new code shown in bold.

Listing A.37: sub-rows-cols.sh

```
#-----------------------------------------
# 1) remove blank lines
# 2) remove line feeds
# 3) print a LF after every fourth field
# 4) remove trailing ',' from each row
#-----------------------------------------
```

```
inputfile="mixed-data.txt"

grep -v "^$" $inputfile |awk -F","
'{printf("%s",$0)}' | awk '
BEGIN { columnCount = 4 }
{
  for(i=1; i<=NF; i++) {
    printf("%s ", $i)
    if( i % columnCount == 0) { print "" }
  }
}' > temp-columns

# 4) remove trailing ',' from output:
cat temp-columns | sed 's/, $//' | sed 's/$ //' >
temp-columns2

cat temp-columns2 | awk '
BEGIN { rowCount = 2; currRow = 0 }
{
  if(currRow % rowCount == 0) { print $1, $3 }
  ++currRow
}' > temp-columns3

cat temp-columns3 | sed 's/,$//' | sed 's/ $//' >
$outputfile
```

Listing A.37 contains a new block of code that redirects the output of step #4 to a temporary file `temp-columns2`, whose contents are processed by another `awk` command in the last section of Listing A.37. Notice that that `awk` command contains a `BEGIN` block that initializes the variables `rowCount` and `currRow` with the values 2 and 0, respectively.

The main block prints columns 1 and 3 of the current line if the current row number is even, and then the value of `currRow` is then incremented. The output of this `awk` command is redirected to yet another temporary file that is the input to the final code snippet, which uses the `cat` command and two occurrences of the `sed` command to remove a trailing “,” and a trailing space, as shown here:

```
cat temp-columns3 | sed 's/,$//' | sed 's/ $//' >
$outputfile
```

There are other ways to perform the functionality in Listing A.37, and the main purpose is to show you different techniques for combining various bash commands.

COUNTING WORD FREQUENCY IN DATASETS

Listing A.38 displays the content of `WordCounts1.sh` that illustrates how to count the frequency of words in a file.

Listing A.38: WordCounts1.sh

```
awk '
# Print list of word frequencies
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}
END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}
' columns2.txt
```

Listing A.38 contains a block of code that processes the lines in `columns2.txt`. Each time that a word (of a line) is encountered, the code increments the number of occurrences of that word in the hash table `freq`. The `END` block contains a `for` loop that displays the number of occurrences of each word in `columns2.txt`.

The output from Listing A.38 is here:

```
two       3
one       3
three     3
six       1
four      3
five      2
```

Listing A.39 displays the content of `WordCounts2.sh` that performs a case-insensitive word count.

Listing A.39: WordCounts2.sh

```
awk '
{
    # convert everything to lower case
    $0 = tolower($0)

    # remove punctuation
   #gsub(/[^[:alnum:]_[:blank:]]/, "", $0)

    for(i=1; i<=NF; i++) {
       freq[$i]++
    }
}
END {
    for(word in freq) {
       printf "%s\t%d\n", word, freq[word]
    }
}
' columns4.txt
```

Listing A.39 contains almost identical code to that in Listing A.38, with the addition of the following code snippet that converts the text in each input line to lowercase letters, as shown here:

```
$0 = tolower($0)
```

Listing A.40 displays the contents of `columns4.txt`.

Listing A.40: columns4.txt

```
123 ONE TWO
456 three four
ONE TWO THREE FOUR
five 123 six
```

```
one two three
four five
```

The output from launching Listing A.39 with `columns4.txt` is here:

```
456      1
two      3
one      3
three    3
six      1
123      2
four     3
five     2
```

DISPLAYING ONLY "PURE" WORDS IN A DATASET

For simplicity, let's work with a text string so we can see the intermediate results as we work toward the solution.

Listing A.41 displays the contents of `onlywords.sh` that contains three `awk` commands for displaying the words, integers, and alphanumeric strings, respectively, in a text string.

Listing A.41: onlywords.sh

```
x="ghi abc Ghi 123 #def5 123z"

echo "Only words:"
echo $x |tr -s ' ' '\n' | awk -F" " '
{
  if($0 ~ /^[a-zA-Z]+$/) { print $0 }
}
' | sort | uniq
echo

echo "Only integers:"
echo $x |tr -s ' ' '\n' | awk -F" " '
{
  if($0 ~ /^[0-9]+$/) { print $0 }
}
' | sort | uniq
echo

echo "Only alphanumeric words:"
echo $x |tr -s ' ' '\n' | awk -F" " '
{
  if($0 ~ /^[0-9a-zA-Z]+$/) { print $0 }
}
' | sort | uniq
echo
```

Listing A.41 starts by initializing the variable `x`:

```
x="ghi abc Ghi 123 #def5 123z"
```

The next step is to split `x` into words:

```
echo $x |tr -s ' ' '\n'
```

The output is here:

```
ghi
abc
Ghi
123
```

```
#def5
123z
```

The third step is to invoke `awk` and check for words that match the regular expression `^[a-zA-Z]+`, which matches any string consisting of one or more uppercase and/or lowercase letters (and nothing else):

```
if($0 ~ /^[a-zA-Z]+$/) { print $0 }
```

The output is here:

```
ghi
abc
Ghi
```

Finally, if you also want to sort the output and print only the unique words, redirect the output from the `awk` command to the `sort` command and the `uniq` command.

The second `awk` command uses the regular expression `^[0-9]+` to check for integers and the third `awk` command uses the regular expression `^[0-9a-zA-Z]+` to check for alphanumeric words. The output from launching Listing A.37 is here:

```
Only words:
Ghi
abc
ghi

Only integers:
123

Only alphanumeric words:
123
```

```
123z
Ghi
abc
ghi
```

Now you can replace the variable `x` with a dataset to retrieve only alphabetic strings from that dataset.

WORKING WITH MULTI-LINE RECORDS IN AWK

Listing A.42 displays the contents of `employee.txt` and Listing A.43 displays the contents of `Employees.sh` that illustrates how to concatenate text lines in a file.

Listing A.42: employees.txt

```
Name: Jane Edwards
EmpId: 12345
Address: 123 Main Street Chicago Illinois

Name: John Smith
EmpId: 23456
Address: 432 Lombard Avenue SF California
```

Listing A.43: employees.sh

```
inputfile="employees.txt"
outputfile="employees2.txt"
awk '
{
  if($0 ~ /^Name:/) {
    x = substr($0,8) ","
    next
  }

  if( $0 ~ /^Empid:/) {
   #skip the Empid data row
   #x = x substr($0,7) ","
    next
  }

  if($0 ~ /^Address:/) {
    x = x substr($0,9)
    print x
  }
}
' < $inputfile > $outputfile
```

The output from launching the code in Listing A.43 is here:

```
Jane Edwards, 123 Main Street Chicago Illinois
John Smith, 432 Lombard Avenue SF California
```

Now that you have seen a plethora of `awk` code snippets and shell scripts containing the `awk` command that illustrate various type of tasks that you can perform on files and datasets, you are ready for some use cases. The next section (which is the first use case) shows you how to replace multiple field delimiters with a single delimiter, and the second use case shows you how to manipulate date strings.

A SIMPLE USE CASE

The code sample in this section shows you how to use the `awk` command to split the comma-separated fields in the rows of a dataset, where fields can contain nested quotes of arbitrary depth.

Listing A.44 displays the content of the file `quotes3.csv` that contains a “,” delimiter and multiple quoted fields.

Listing A.44: quotes3.csv

```
field5,field4,field3,"field2,foo,bar",field1,field
6,field7,"fieldZ"
fname1,"fname2,other,stuff",fname3,"fname4,foo,ba
r",fname5
"lname1,a,b","lname2,c,d","lname3,e,f","lname4,fo
o,bar",lname5
```

Listing A.45 displays the content of the file `delim1.sh` that illustrates how to replace the delimiters in `delim1.csv` with a “,” character.

Listing A.45: delim1.sh

```
#inputfile="quotes1.csv"
#inputfile="quotes2.csv"
inputfile="quotes3.csv"

grep -v "^$" $inputfile | awk '
{
  print "LINE #" NR ": " $0
  printf ("-------------------------\n")
  for (i = 0; ++i <= NF;)
    printf "field #%d : %s\n", i, $i
  printf ("\n")
}' FPAT='([^,]+)|("[^"]+" )' < $inputfile
```

The output from launching the shell script in Listing A.44 is here:

```
LINE #1: field5,field4,field3,"field2,foo,bar",fie
ld1,field6,field7,"fieldZ"
-------------------------
field #1 : field5
field #2 : field4
field #3 : field3
field #4 : "field2,foo,bar"
field #5 : field1
field #6 : field6
field #7 : field7
field #8 : "fieldZ"

LINE #2: fname1,"fname2,other,stuff",fname3,"fname
4,foo,bar",fname5
-------------------------
field #1 : fname1
field #2 : "fname2,other,stuff"
field #3 : fname3
field #4 : "fname4,foo,bar"
field #5 : fname5

LINE #3: "lname1,a,b","lname2,c,d","lname3,e,f","l
name4,foo,bar",lname5
-------------------------
field #1 : "lname1,a,b"
field #2 : "lname2,c,d"
field #3 : "lname3,e,f"
field #4 : "lname4,foo,bar"
field #5 : lname5
```

```
LINE #4: "Outer1 "Inner "Inner "Inner C" B" A"
Outer1","XYZ1,c,d","XYZ2lname3,e,f"
-------------------------
field #1 : "Outer1 "Inner "Inner "Inner C" B" A"
Outer1"
field #2 : "XYZ1,c,d"
field #3 : "XYZ2lname3,e,f"

LINE #5:
-------------------------
```

As you can see, the task in this section is easily solved via the `awk` command.

ANOTHER USE CASE

The code sample in this section shows you how to use the `awk` command to reformat the date field in a dataset and change the order of the fields in the new dataset. We have the following input line in the original dataset,

```
Jane,Smith,20140805234658
```

The reformatted line in the output dataset has this format:

```
2014-08-05 23:46:58,Jane,Smith
```

Listing A.46 displays the content of the file `dates2.csv` that contains a “,” delimiter and three fields.

Listing A.46: dates2.csv

```
Jane,Smith,20140805234658
Jack,Jones,20170805234652
Dave,Stone,20160805234655
```

```
John,Smith,20130805234646
Jean,Davis,20140805234649
Thad,Smith,20150805234637
Jack,Pruit,20160805234638
```

Listing A.47 displays the content of `string2date2.sh` that converts the date field to a new format and shifts the new date to the first field.

Listing A.47: string2date2.sh

```
inputfile="dates2.csv"
outputfile="formatteddates2.csv"

rm -f $outputfile; touch $outputfile

for line in `cat $inputfile`
do
  fname=`echo $line |cut -d"," -f1`
  lname=`echo $line |cut -d"," -f2`
  date1=`echo $line |cut -d"," -f3`

  # convert to new date format
  newdate=`echo $date1 | awk '{ print
  substr($0,1,4)"-
"substr($0,5,2)"-"substr($0,7,2)"
"substr($0,9,2)":"substr
($0,11,2)":"substr($0,13,2)}'`

  # append newly formatted row to output file
  echo "${newdate},${fname},${lname}" >>
  $outputfile
done
```

The contents of the new dataset are here:

```
2014-08-05 23:46:58,Jane,Smith
2017-08-05 23:46:52,Jack,Jones
2016-08-05 23:46:55,Dave,Stone
2013-08-05 23:46:46,John,Smith
2014-08-05 23:46:49,Jean,Davis
2015-08-05 23:46:37,Thad,Smith
2016-08-05 23:46:38,Jack,Pruit
```

SUMMARY

This appendix introduced the `awk` command, which is essentially an entire programming language packaged into a single Unix command.

We explored some of its built-in variables as well as conditional logic, `while` loops, and `for` loops to manipulate the rows and columns in datasets. You then saw how to delete lines and merge lines in datasets, as well as how to print the contents of a file as a single line of text.

Next, you learned how to use metacharacters and character sets in `awk` commands. You learned how to perform numeric calculations (such as addition, subtraction, multiplication, and division) in files containing numeric data, as well as some numeric functions that are available in `awk`.

In addition, you saw how to align and delete columns, select a subset of columns, and work with multi-line records in datasets. Finally, you saw simple use cases involving nested quotation marks and date formats in a structured dataset.

At this point, you have all the tools necessary to perform sophisticated data cleansing and processing, and you are strongly encouraged to apply them to some task or problem of interest. The final step of the learning process is working on a real-life application.

OceanofPDF.com

INDEX

A

Accuracy, 170–171
Algorithms, 155
Aligning columns, 344–348
Anomaly, 64
Anomaly detection, 65–70
Attribute selection, 213
AUC curve, 176–181
Awk command, 315–362

B

Balanced accuracy, 170–171
Best-fitting line, 254–255
Binary confusion matrix, 169
Binary data, 10
“Binning” data values, 21–22
Black-box shift detector, 18
Bmi.csv dataset, 197–198
Bokeh, 292–295
Built-in variables, 316–317

C

Carling Median Rule, 49

Categorical data, 12, 32–33, 101–102, 120–125
Categorical features, 217
Caveat regarding accuracy, 171
Character sets, 329–330
Charts, 268–270
Checkerboard, 244–245
Cleaning datasets, 16
Clustering, 209
Collinearity, 1, 25
Color values, 230–231
Column-aligned rows, 349–351
Column dependencies, 15
Column subset, 116–117
Conditional logic, 320–323, 330–331
Confusion matrix, 163–175
Continuous data, 12–13, 20–21
Continuous features, 217
Control statements, 320–323
Convolutional Neural Network (CNN), 4
Correlation, 1, 26
Correlation matrix, 26
Cost-sensitive learning, 73–74
Counting missing data values, 28–29
Counting word frequency, 351–353
Count occurrences of string, 341–342
Covariate drift, 18
CSV files, 80–91, 111–115
Cubed numbers, 231–232
Currency, 125–135

D

Data-centric AI, 4–5
Data classes, 40–42
Data cleaning, 5, 27–43, 219–222
Data cleaning tasks, 120
Data dealing, 6
Data deduplication, 30, 95–96
Data drift, 1, 17–18
Data governance, 4
Data inconsistency, 33, 102
Data leakage, 1, 18–19
Data normalization, 40, 117–120, 156–158
Data point, 6
Data preprocessing, 15–16
Data quality, 4
Data quality assurance, 4
Data sampling techniques, 71–72
Datasets, 6–8, 19–23, 77–153, 323–328, 344–356
Data splits, 76
Data stewardship, 4
Data transformation, 151–152
Data types, 1, 10–17
Data validation, 43
Data values, uniformity of, 30–32
Data visualization, 205–208, 225–226
Data wrangling, 5, 150–152
Dates, 135–145
DBSCAN clustering method, 68–69
3D charts, 264–265
DecisionTreeClassifier, 195

Decision trees, 210–212
DeepChecks, 43
Deep learning, 70
Deleting alternate lines, 323–324
Dependency types, 23–27
Dependent column, 15
Diabetes.csv dataset, 198–200
Differential privacy, 19
Digits dataset, 297–301
Dimensionality reduction, 218
Discrete data, 13, 20–21
Discrimination threshold, 27
Display attribute values, 228–230
Doane’s formula, 22
Domain classifier, 18
Dotted grid, 238–240
Drift, 17–18
Drop redundant columns, 30, 94
Duplicate rows, 96–99
Dynamic dataset, 7

E

Embedded methods, 215
Error matrix, 169
Exploratory data analysis (EDA), 1–6, 205–208
Extract, load, and transform (ELT), 5–6
Extract, transform, and load (ETL), 5–6

F

False discovery rate (FDR), 175

False omission rate (FOR), 175
False positive rate (FPR), 174
Feature, 6
Feature drift, 18
Feature engineering, 212–213
Feature extraction, 212, 218–219
Feature hashing, 219
Feature selection, 213–218
Filter methods, 214–215
Fixed number of columns, 342–344
Fraud detection, 63–64
Freedman-Diaconis' Choice, 23

G

GaussianNB, 195, 196
GGPLOT, 254–255
Good correlation value, 26–27
Graphs, 268–270

H

Heat maps, 207–208, 259–260
Heteroskedasticity, 24
Histogram, 205–207, 250–251
Histogram-based Outlier Score (HBOS), 67
Homoskedasticity, 1, 24–25
Horizontal lines, 233–234
Hot-deck imputation, 38–39, 107
Hybrid methods, 215

I

Imbalanced datasets, 70–76
Imputation, 27–43, 91–108
Inconsistent categorical data, 121
Incorrectly scaled values, 48
Input drift, 18
Integer-based ordinal data, 11
Interval data, 12
Iris dataset, 274–275, 301–311
Isolation forest, 69–70
Isolation number, 69

K

KNN-based model, 192–194, 197–200
Kullback-Leibler (KL) divergence, 219

L

Labeled data, 220
Labeled vertices, 237–238
Large datasets, 221
Linear regression, 24–25
Lines vertices, 237–238
Loading images, 243–244
Localization of data, 135
Local Outlier Factor (LOF), 66, 67

M

Machine learning, 219–222
Matching, 38–39, 107

Matplotlib, 223–314
Matplotlib styles, 227–228
Mean value imputation, 33–35, 102–104
Median Absolute Deviation (MAD), 49
Merging lines, 324–328
Metacharacters, 329–330
Missing At Random (MAR), 8–9
Missing Completely At Random (MCAR), 8–9
Missing data, 8–10, 78–79, 101
Missing Not At Random (MNAR), 8–9
Model(s), 153–222
Model-centric AI, 4–5
Model drift, 17–18
Model selection, 19–23
Modified Z-score, 49
Multicollinearity, 25
Multi-line records, 356–357
Multiple imputation, 38
Multiple rows, 346–348
Multi-row records, 111–115

N

Negative predictive value (NPV), 172–173, 175
Noisy data, 72–73
Nominal data, 10–11
Normalization, 40, 117–120, 156
Normalized confusion matrix, 165–166
Numeric functions, 334–337
Numeric Outlier (IQR), 68
Numeric values, 121–123

NumPy, 49–53

O

One-hot encoding, 32, 122, 123–125, 217
One-line awk commands, 337–338
Ordinal data, 11
Outliers, 46–63

P

Pandas code, 2, 45, 54–61
Pandas data frame, 80–91, 283–283
Parallel slanted lines, 235–237
Partitioned datasets, 190–192
Pie chart, 258
Plot bar charts, 255–257
Plotting financial data, 265–267
Plotting multiple lines, 248–249
PNG file, 260–261
Postfix arithmetic operators, 332–334
Precision, 172–174
Prevalence, 172–173
Principal Component Analysis (PCA), 7
Printf() statement, 318–319
Printing lines, 330–331
Probabilistic graphical model (PGM), 66
Programmatic binning techniques, 22–23
Python, 2, 45
Python-based code samples, 1–2
Python code sample, 166–168
Python Outlier Detection (PyOD), 67–68

Q

Qualitative data, 13–14
Quantitative data, 13
Quantitative data analysis techniques, 3
Quoted fields, 145–149

R

RandomForestClassifier, 195, 196
Randomized data points, 246–247
Random oversampling, 71
Random resampling, 71
Random undersampling technique, 71
Random value imputation, 36–38, 105–107
Range dates, find out of, 136–139
Ratio data, 12
Rebalancing datasets, 75
Recall, 172–174
Receiving Operator Characteristics (ROC) curve, 176–181
Regression, 209
Resource bundle, 16
Robust standardization, 162
Row subrange, 116–117

S

Scaling data, 154–163
Scaling numeric data, 159–161
Scikit-learn, 61–63, 296–311
Scoring rule, 180
Seaborn, 223–314

Seaborn built-in datasets, 273
Seaborn dataset names, 272–273
Seaborn heat maps, 286–288
Seaborn pair plots, 288–292
Set of line segments, 247–248
Short awk scripts, 338–340
Single column CSV files, 80–86
Skewed datasets, 108–111
Skimpy, 263–264
Slanted lines, 234–235
Specificity, 172–173, 176
Splitting data, 155–163
Splitting file names, 331–332
SQLITE3, 252–253, 268–270
Standardization, 156, 160
StandardScaler Class, 161–162
Static dataset, 7
Statistical data, 14–15
Sturge’s rule, 23
SVC, 195, 196
SweetViz, 262–263
Switch date formats, 143–145
Synthetic data labels, 220
Synthetic Minority Oversampling Technique (SMOTE), 72, 79, 149–150, 200–205

T

Text file, 348–349
Text string, 340
Titanic CSV file, 116–117

Titanic dataset, 181–189, 200–205, 275–283
TOC curve, 180
Training classifiers, 189–190
Trigonometric functions, 249–250
True positive rate (TPR), 174–175
Tukey's boxplot, 49
Two-column CSV files, 86–91
Two lines and a legend, 242–243
Type I errors, 169–170
Type II errors, 169–170

U

Uniformity of data values, 30–32, 99–100
Unique dates, 141–143
Units of measure, 159
Unlabeled data, 220
Unsupervised feature selection techniques, 215

V

Variable selection, 213
Variable subset selection, 213
Variance inflation factor (VIF), 1, 25

W

Wikipedia, 3
Wine.csv dataset, 192–196
Wrapper methods, 215

X

XGBoost, 49

Z

Zero value, 39–40, 107–108
Z-scores, 59–61, 68

OceanofPDF.com